■ **Tip** You need to run this command only the first time you create a Blazor WebAssembly project. Once the template is installed, you can skip this step and jump to the command shown in Listing 37-6.

Listing 37-5. Installing the Blazor WebAssembly Project Template

```
dotnet new -i Microsoft.AspNetCore.Blazor.Templates::3.1.0-preview4.19579.2
```

The output from this command is confusing and can leave you with the impression that the installation has failed. The command shows a list of the installed templates, and you will know the command has succeeded if you see this entry in the list:

```
...
Blazor WebAssembly App    blazorwasm    [C#]    Web/Blazor/WebAssembly
...
```

Next, use the PowerShell command prompt to run the commands shown in Listing 37-6 from within the Advanced project folder (the folder that contains the Advanced.csproj file).

Listing 37-6. Creating the Blazor WebAssembly Project

```
dotnet new blazorwasm -o ../BlazorWebAssembly
dotnet add ../BlazorWebAssembly reference ../DataModel
```

These commands create a Blazor WebAssembly project named BlazorWebAssembly and add a reference to the DataModel project, which makes the Person, Department, and Location classes available.

Preparing the ASP.NET Core Project

Use the PowerShell command prompt to run the commands shown in Listing 37-7 in the Advanced project folder.

Listing 37-7. Preparing the Advanced Project

```
dotnet add reference ../DataModel ../BlazorWebAssembly

dotnet add package Microsoft.AspNetCore.Blazor.Server --version 3.1.0-preview4.19579.2
```

These commands create references to the other projects so that the data model classes and the components in the Blazor WebAssembly project can be used.

Adding the Solution References

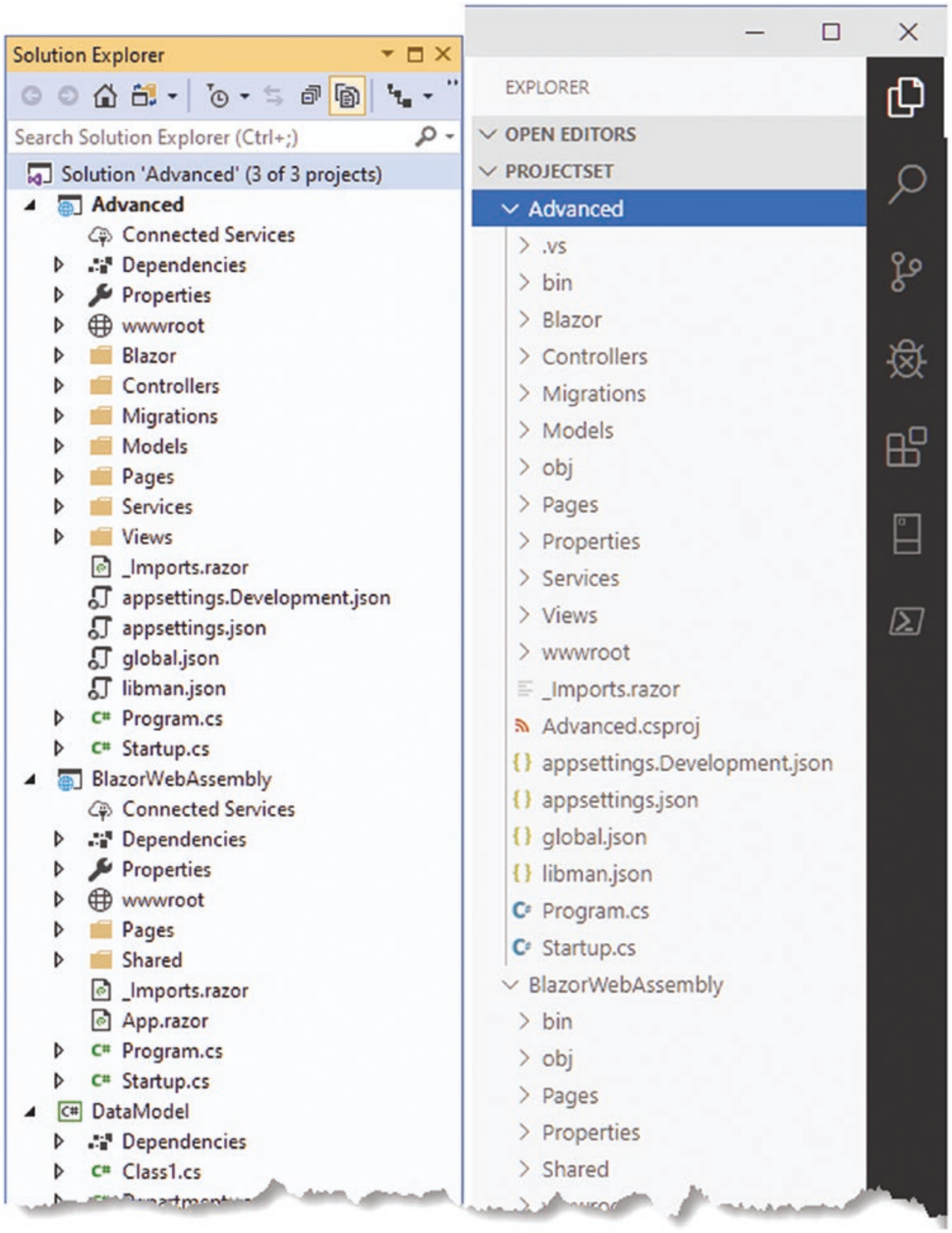
Run the command shown in Listing 37-8 in the Advanced folder to add references to the new project to the solution file.

Listing 37-8. Adding Solution References

```
dotnet sln add ../DataModel ../BlazorWebAssembly
```

Opening the Projects

Once you have set up all three projects, start Visual Studio or Visual Studio Code. If you are using Visual Studio, open the `Advanced.sln` file in the `Advanced` folder. All three projects are open for editing, as shown in Figure 37-2. If you are using Visual Studio Code, open the folder that contains all three projects, as shown in Figure 37-2.



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Advanced' (3 of 3 projects)
  Advanced
    Connected Services
    Dependencies
    Properties
    wwwroot
    Blazor
    Controllers
    Migrations
    Models
    Pages
    Services
    Views
    _Imports.razor
    appsettings.Development.json
    appsettings.json
    global.json
    libman.json
    Program.cs
    Startup.cs
  BlazorWebAssembly
    Connected Services
    Dependencies
    Properties
    wwwroot
    Pages
    Shared
    _Imports.razor
    App.razor
    Program.cs
    Startup.cs
  DataModel
    Dependencies
    Class1.cs
```

```
EXPLORER
OPEN EDITORS
PROJECTSET
  Advanced
    .vs
    bin
    Blazor
    Controllers
    Migrations
    Models
    obj
    Pages
    Properties
    Services
    Views
    wwwroot
    _Imports.razor
    Advanced.csproj
    appsettings.Development.json
    appsettings.json
    global.json
    libman.json
    Program.cs
    Startup.cs
  BlazorWebAssembly
    bin
    obj
    Pages
    Properties
    Shared
```

Figure 37-2. *Opening the three projects*

Completing the Blazor WebAssembly Configuration

The next step is to configure the ASP.NET Core project so that it can deliver the contents of the Blazor WebAssembly project to clients. Add the statements shown in Listing 37-9 to the `Startup.cs` file in the `Advanced` folder.

■ **Caution** It is important to pay close attention to which files you are editing. Files with the same name exist in multiple projects, and if you don't follow the examples closely, you won't end up with a working application. Future versions of Blazor may be easier to work with, but for the moment, the details are important.

Listing 37-9. Configuring the Application in the Startup.cs File in the Advanced Project

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Advanced.Models;
using Microsoft.AspNetCore.ResponseCompression;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddServerSideBlazor();
            services.AddSingleton<Services.ToggleService>();

            services.AddResponseCompression(opts => {
                opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
                    new[] { "application/octet-stream" });
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("controllers",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
                endpoints.MapBlazorHub();

                endpoints.MapFallbackToClientSideBlazor<BlazorWebAssembly.Startup>
                    ("/webassembly/{*path:nonfile}", "index.html");

                endpoints.MapFallbackToPage("/_Host");
            });

            app.Map("/webassembly", opts =>
                opts.UseClientSideBlazorFiles<BlazorWebAssembly.Startup>());

            SeedData.SeedDatabase(context);
        }
    }
}
```

These statements enable response compression, which Blazor WebAssembly requires, and configure the ASP.NET Core request pipeline so that requests for `/webassembly` are handled by Blazor WebAssembly using the contents of the `BlazorWebAssembly` project.

Setting the Base URL

The final step is to modify the HTML file that will be used to respond to requests for the `/webassembly` URL. Apply the change shown in Listing 37-10 to the `index.html` file in the `wwwroot` folder of the `BlazorWebAssembly` folder.

■ **Caution** Make sure there are forward-slash (`/`) characters before and after `webassembly` in the `href` attribute of the base element. If you omit either character, then Blazor WebAssembly will not work.

Listing 37-10. Setting the URL in the index.html File in the wwwroot Folder of the BlazorWebAssembly Project

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>BlazorWebAssembly</title>
    <base href="/webassembly/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
</head>

<body>
    <app>Loading...</app>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">x</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

The base element sets the URL from which all relative URLs in the document are defined and is required for the correct operation of the Blazor WebAssembly routing system.

Testing the Placeholder Components

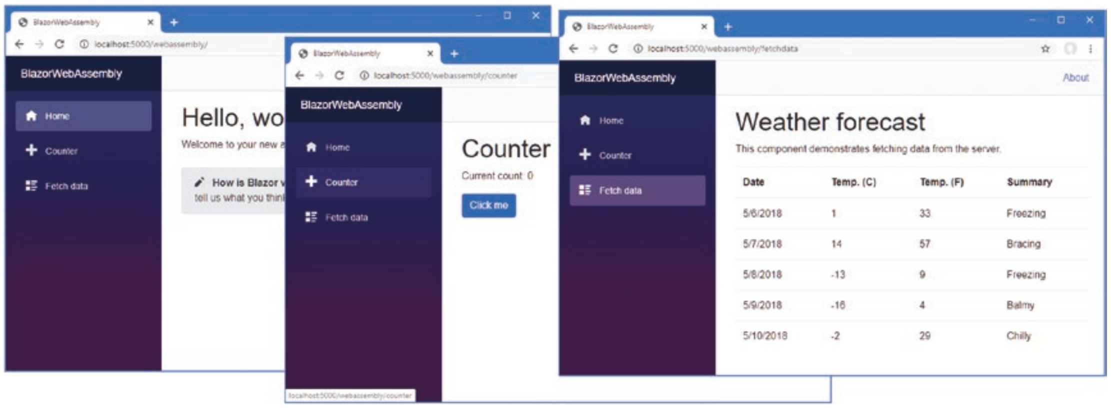
Start ASP.NET Core by selecting Start Without Debugging or Run Without Debugging from the Debug menu. If you prefer to use the command prompt, run the command shown in Listing 37-11 in the Advanced project folder.

Listing 37-11. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/webassembly`, and you will see the placeholder content added by the template used to create the BlazorWebAssembly project.

Using the PowerShell command prompt, run the following commands from within the `Advanced` project folder. Click the Counter and Fetch Data links, and you will see different content displayed, as shown in Figure 37-3.



Date	Temp. (C)	Temp. (F)	Summary
5/6/2018	1	33	Freezing
5/7/2018	14	57	Bracing
5/8/2018	-13	9	Freezing
5/9/2018	-16	4	Balmy
5/10/2018	-2	29	Chilly

Figure 37-3. *The Blazor WebAssembly placeholder content*

Creating a Blazor WebAssembly Component

Blazor WebAssembly uses the same approach as Blazor Server, relying on components as building blocks for applications, connected through the routing system, and displaying common content through layouts. In this section, I show how to create a Razor Component that works with Blazor WebAssembly, and then I'll re-create the simple forms application from Chapter 36.

Importing the Data Model Namespace

The components I will create in this chapter all use the classes in the shared DataModel project. Rather than add `@using` expressions to each component, add the namespace for the data model classes to the `_Imports.razor` file in the root folder of the BlazorWebAssembly project, as shown in Listing 37-12.

Listing 37-12. Adding a Namespace in the _Imports.razor File in the BlazorWebAssembly Project

```
@using System.Net.Http
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
```

```
@using BlazorWebAssembly
@using BlazorWebAssembly.Shared
@using Advanced.Models
```

Notice that although I moved the model classes to the `DataModel` project, I have specified the `Advanced.Models` namespace. This is because the class files I moved all have `namespace` declarations that specify `Advanced.Models`, which means that moving the files hasn't changed the namespace in which the classes exist.

Creating a Component

In earlier chapters, I defined my Razor Components in a `Blazor` folder to keep the new content separate from the other parts of ASP.NET Core. There is only Blazor content in the BlazorWebAssembly project, so I am going to follow the convention adopted by the project template and use the `Pages` and `Shared` folders.

Add a Razor Component named `List.razor` to the `Pages` folder of the BlazorWebAssembly project and add the content shown in Listing 37-13.

Listing 37-13. The Contents of the List.razor File in the Pages Folder of the BlazorWebAssembly Project

```
@page "/forms"
@page "/forms/list"

<h5 class="bg-primary text-white text-center p-2">People (WebAssembly)</h5>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>ID</th><th>Name</th><th>Dept</th><th>Location</th><th></th>
        </tr>
    </thead>
    <tbody>
        @if  (People.Count() == 0) {
            <tr><th colspan="5" class="p-4 text-center">Loading Data...</th></tr>
        } else {
            @foreach (Person p in People) {
                <tr>
                    <td>@p.PersonId</td>
                    <td>@p.Surname, @p.Firstname</td>
                    <td>@p.Department.Name</td>
                    <td>@p.Location.City</td>
                    <td class="text-center">
                        <NavLink class="btn btn-sm btn-info"
                               href="@GetDetailsUrl(p.PersonId)">
                            Details
                        </NavLink>
                        <NavLink class="btn btn-sm btn-warning"
                               href="@GetEditUrl(p.PersonId)">
                            Edit
                        </NavLink>
                        <button class="btn btn-sm btn-danger"
                                @onclick="@(() => HandleDelete(p))">
                            Delete
                        </button>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

```
<NavLink class="btn btn-primary" href="forms/create">Create</NavLink>

@code {

    [Inject]
    public HttpClient Http { get; set; }

    public Person[] People { get; set; } = Array.Empty<Person>();

    protected async override Task OnInitializedAsync() {
        await UpdateData();
    }

    private async Task UpdateData() {
        People = await Http.GetJsonAsync<Person[]>("/api/people");
    }

    string GetEditUrl(long id) => $"forms/edit/{id}";
    string GetDetailsUrl(long id) => $"forms/details/{id}";

    public async Task HandleDelete(Person p) {
        HttpResponseMessage resp =
            await Http.DeleteAsync($"/api/people/{p.PersonId}");
        if (resp.IsSuccessStatusCode) {
            await UpdateData();
        }
    }
}
```

If you compare this component with the Blazor Server equivalent from Chapter 36, you will see that they are largely the same. Both types of Blazor use the same set of core features, which is why the content uses the same Razor directives, handles events with the `@onclick` attributes, and uses the same `@code` section for C# statements. A Blazor WebAssembly component is compiled into a C# class, just like its Blazor Server counterpart. The key difference is, of course, that the C# class that is generated is executed in the browser—and that's the reason for the differences from the component in Chapter 36.

Navigating in a Blazor WebAssembly Component

Notice that the URLs that are used for navigation are expressed without a leading forward-slash character, like this:

```
...
<NavLink class="btn btn-primary" href="forms/create">Create</NavLink>
...
```

The root URL for the application was specified using the `base` element in Listing 37-13, and using relative URLs ensures that navigation is performed relative to the root. In this case, the relative `forms/create` URL is combined with the `/webassembly/` root specified by the `base` element, and navigation will be to `/webassembly/forms/create`. Including a leading forward slash would navigate to `/forms/create` instead, which is outside the set of URLs that are being managed by the Blazor WebAssembly part of the application. This change is required only for navigation URLs. URLs specified with the `@page` directive, for example, are not affected.

Getting Data in a Blazor WebAssembly Component

The biggest change is that Blazor WebAssembly can't use Entity Framework Core. Although the runtime may be able to execute the Entity Framework Core classes, the browser restricts WebAssembly applications to HTTP requests, preventing the use of SQL. To get data, Blazor WebAssembly applications consume web services, which is why I added the API controller to the Advanced project at the start of the chapter.

As part of the Blazor WebAssembly application startup, a service is created for the `HttpClient` class, which components can receive using the standard dependency injection features. The `List` component receives an `HttpClient` component through a property that has been decorated with the `Inject` attribute, like this:

```
...
[Inject]
public HttpClient Http { get; set; }
...
```

The `HttpClient` class provides the methods described in Table 37-2 to send HTTP requests.

Table 37-2. *The Methods Defined by the HttpClient Class*

Name	Description
`GetAsync(url)`	This method sends an HTTP GET request.
`PostAsync(url, data)`	This method sends an HTTP POST request.
`PutAsync(url, data)`	This method sends an HTTP PUT request.
`PatchAynC(url, data)`	This method sends an HTTP PATCH request.
`DeleteAsync(url)`	This method sends an HTTP DELETE request.
`SendAsync(request)`	This method sends an HTTP, configured using an `HttpRequestMessage` object.

The methods in Table 37-2 return a `Task<HttpResponseMessage>` result, which describes the response received from the HTTP server to the asynchronous request. Table 37-3 shows the most useful `HttpResponseMessage` properties.

Table 37-3. *Useful HttpClient Properties*

Name	Description
`Content`	This property returns the content returned by the server.
`HttpResponseHeaders`	This property returns the response headers.
`StatusCode`	This property returns the response status code.
`IsSuccessStatusCode`	This property returns `true` if the response status code is between 200 and 299, indicating a successful request.

The `List` component uses the `DeleteAsync` methods to ask the web service to delete objects when the user clicks a Delete button.

```
...
HttpResponseMessage resp =
    await Http.DeleteAsync($"/api/people/{p.PersonId}");
    if (resp.IsSuccessStatusCode) {
        await UpdateData();
    }
}
...
```

These methods are useful when you don't need to work with the data the web service sends back, such as in this situation where I check to see only if the DELETE request has been successful. Notice that I specify the path for the request URL only when using the `HttpClient` service because the web service is available using the same scheme, host, and port as the application.

For operations where the web service returns data, the extension methods for the `HttpClient` class described in Table 37-4 are more useful. These methods serialize data into JSON so it can be sent to the server and parse JSON responses into C# objects. For requests that return no result, the generic type argument can be omitted.

Table 37-4. *The HttpClient Extension Methods*

Name	Description
`GetJsonAsync<T>(url)`	This method sends an HTTP GET request and parses the response to type `T`.
`PostJsonAsync<T>(url, data)`	This method sends an HTTP POST request and parses the response to type `T`.
`PutJsonAsync<T>(url, data)`	This method sends an HTTP PUT request and parses the response to type `T`.
`SendJsonAsync<T>(method, url, data)`	This method sends an HTTP request using the specified method and parses the response to type `T`.

The `List` component uses the `GetJsonAsync<T>` method to request data from the web service.

```
...
private async Task UpdateData() {
    People = await Http.GetJsonAsync<Person[]>("/api/people");
}
...
```

Setting the generic type argument to `Person[]` tells `HttpClient` to parse the response into an array of `Person` objects.

■ **Note** The `HttpClient` class doesn't present any scope or lifecycle issues and sends requests only when one of the methods described in Table 37-2 or Table 37-4 is invoked. Some thought is required, however, about when to request new data. In this example, I requery the web service after an object has been deleted, rather than simply remove the object from the data that was requested when the component was initialized. This may not be suitable for all applications because it will reflect any changes to the database that have been made by other users.

Creating a Layout

The template used to create the Blazor WebAssembly project includes a layout that presents the navigation features for the placeholder content. I don't want these navigation features, so the first step is to create a new layout. Add a Razor Component named `EmptyLayout.razor` in the `Shared` folder of the BlazorWebAssembly project with the content shown in Listing 37-14.

Listing 37-14. The EmptyLayout.razor File in the Shared Folder of the BlazorWebAssembly Project

```
@inherits LayoutComponentBase

<div class="m-2">
    @Body
</div>
```

I could apply the new layout with `@layout` expressions, as I did in Chapter 36, but I am going to use this layout as the default by changing the routing configuration, which is defined in the `App.razor` file in the BlazorWebAssembly project, as shown in Listing 37-15.

Listing 37-15. Applying the Layout in the App.razor File in the BlazorWebAssembly Project

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(EmptyLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(EmptyLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

Chapter 35 describes the Router, RouteView, Found, and NotFound components.

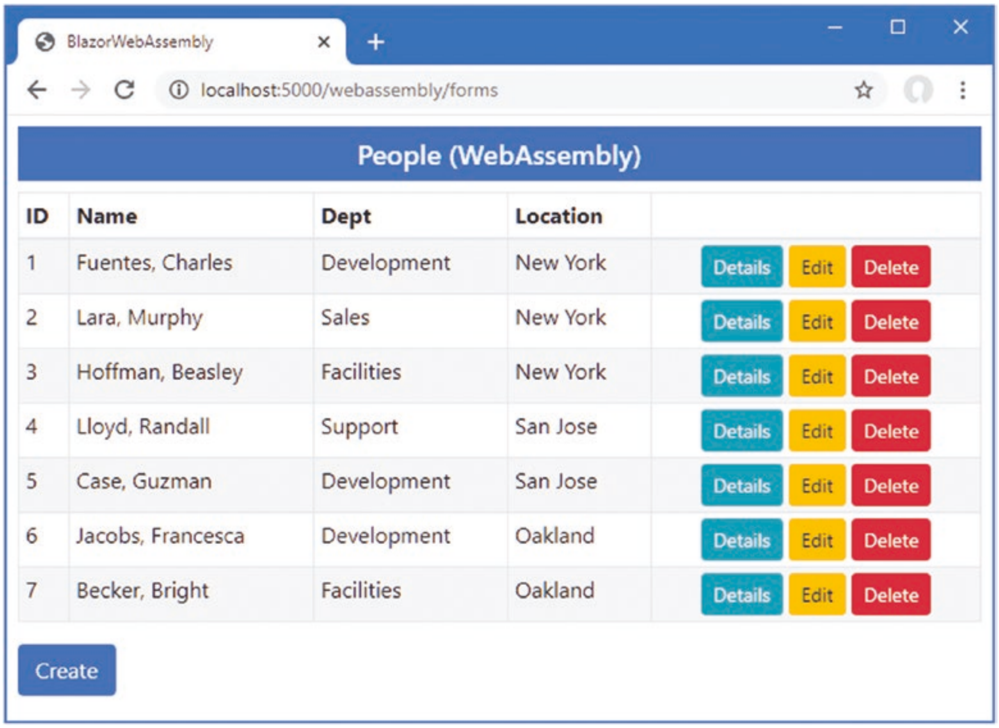
Defining CSS Styles

The template created the Blazor WebAssembly project with its own copy of the Bootstrap CSS framework and with an additional stylesheet that combines the styles required to configure the Blazor WebAssembly error and validation elements and manage the layout of the application. Replace the link elements in the HTML file as shown in Listing 37-16 and apply styles directly to the error element. This has the effect of removing the styles used by the Microsoft layout and using the Bootstrap CSS stylesheet that was added to the Advanced project.

Listing 37-16. Modifying the index.html File in the wwwroot Folder in the BlazorWebAssembly Project

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>BlazorWebAssembly</title>
    <base href="/webassembly/" />
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>

<body>
    <app>Loading...</app>

    <div id="blazor-error-ui"
              class="text-center bg-danger h6 text-white p-2 fixed-top w-100"
                style="display:none">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">x</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

To see the new component, restart ASP.NET Core and request http://localhost:5000/webassembly/forms, which will produce the response shown in Figure 37-4.



People (WebAssembly)

ID	Name	Dept	Location	
1	Fuentes, Charles	Development	New York	Details Edit Delete
2	Lara, Murphy	Sales	New York	Details Edit Delete
3	Hoffman, Beasley	Facilities	New York	Details Edit Delete
4	Lloyd, Randall	Support	San Jose	Details Edit Delete
5	Case, Guzman	Development	San Jose	Details Edit Delete
6	Jacobs, Francesca	Development	Oakland	Details Edit Delete
7	Becker, Bright	Facilities	Oakland	Details Edit Delete

Figure 37-4. *A Blazor WebAssembly component*

Blazor WebAssembly components follow the standard Blazor lifecycle, and the component displays the data it receives from the web service.

Completing the Blazor WebAssembly Form Application

Only the Delete button displayed by the `List` component works currently. In the sections that follow, I complete the Blazor WebAssembly form application by creating additional components.

Creating the Details Component

Add a Razor Component named `Details.razor` to the `Pages` folder of the BlazorWebAssembly project with the content shown in Listing 37-17.

Listing 37-17. The Contents of the Details.razor File in the Pages Folder of the BlazorWebAssembly Project

```
@page "/forms/details/{id:long}"

<h4 class="bg-info text-center text-white p-2">Details (WebAssembly)</h4>

<div class="form-group">
    <label>ID</label>
    <input class="form-control" value="@PersonData.PersonId" disabled />
</div>
<div class="form-group">
    <label>Firstname</label>
    <input class="form-control" value="@PersonData.Firstname" disabled />
</div>
<div class="form-group">
    <label>Surname</label>
    <input class="form-control" value="@PersonData.Surname" disabled />
</div>
```

```
<div class="form-group">
    <label>Department</label>
    <input class="form-control" value="@PersonData.Department?.Name" disabled />
</div>
<div class="form-group">
    <label>Location</label>
    <input class="form-control"
           value="@($"{PersonData.Location?.City}, {PersonData.Location?.State}")"
           disabled />
</div>
<div class="text-center">
    <NavLink class="btn btn-info" href="@EditUrl">Edit</NavLink>
    <NavLink class="btn btn-secondary" href="forms">Back</NavLink>
</div>

@code {

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Inject]
    public HttpClient Http { get; set; }

    [Parameter]
    public long Id { get; set; }

    public Person PersonData { get; set; } = new Person();

    protected async override Task OnParametersSetAsync() {
        PersonData = await Http.GetJsonAsync<Person>($"/api/people/{Id}");
    }

    public string EditUrl => $"forms/edit/{Id}";
}
```

The `Details` component has only two differences from its Blazor Server counterpart, following the pattern established by the `List` component: the data is obtained through the `HttpClient` service, and navigation targets are expressed using relative URLs. In all other regards, such as obtaining parameters from routing data, Blazor WebAssembly works just the same way as Blazor Server.

Creating the Editor Component

To complete the forms application, add a Razor Component named `Editor.razor` to the `Pages` folder of the `BlazorWebAssembly` project with the content shown in Listing 37-18.

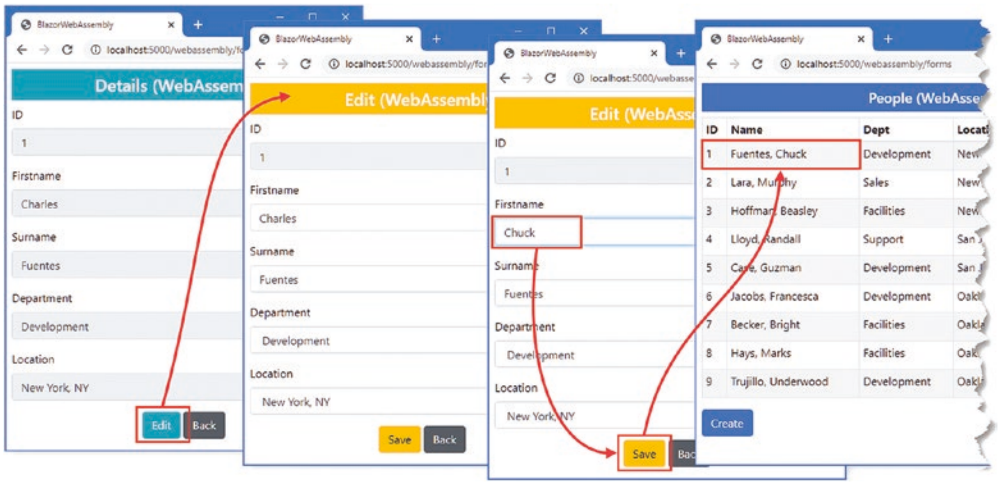
Listing 37-18. The Contents of the Editor.razor File in the Pages Folder of the BlazorWebAssembly Project

```
@page "/forms/edit/{id:long}"
@page "/forms/create"

<link href="/blazorValidation.css" rel="stylesheet" />

<h4 class="bg-@Theme text-center text-white p-2">@Mode (WebAssembly)</h4>

<EditForm Model="PersonData" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    @if (Mode == "Edit") {
        <div class="form-group">
            <label>ID</label>
            <InputNumber class="form-control"
                @bind-Value="PersonData.PersonId" readonly />
        </div>
    }
    <div class="form-group">
        <label>Firstname</label>
        <ValidationMessage For="@(() => PersonData.Firstname)" />
        <InputText class="form-control" @bind-Value="PersonData.Firstname" />
    </div>
    <div class="form-group">
        <label>Surname</label>
        <ValidationMessage For="@(() => PersonData.Surname)" />
        <InputText class="form-control" @bind-Value="PersonData.Surname" />
    </div>
    <div class="form-group">
        <label>Department</label>
        <ValidationMessage For="@(() => PersonData.DepartmentId)" />
        <select @bind="PersonData.DepartmentId" class="form-control">
            <option selected disabled value="0">Choose a Department</option>
            @foreach (var kvp in Departments) {
                <option value="@kvp.Value">@kvp.Key</option>
            }
        </select>
    </div>
    <div class="form-group">
        <label>Location</label>
        <ValidationMessage For="@(() => PersonData.LocationId)" />
        <select @bind="PersonData.LocationId" class="form-control">
            <option selected disabled value="0">Choose a Location</option>
            @foreach (var kvp in Locations) {
                <option value="@kvp.Value">@kvp.Key</option>
            }
        </select>
    </div>
    <div class="text-center">
        <button type="submit" class="btn btn-@Theme">Save</button>
        <NavLink class="btn btn-secondary" href="forms">Back</NavLink>
    </div>
</EditForm>

@code {

    [Inject]
    public HttpClient Http { get; set; }

    [Inject]
    public NavigationManager NavManager { get; set; }

    [Parameter]
    public long Id { get; set; }

    public Person PersonData { get; set; } = new Person();

    public IDictionary<string, long> Departments { get; set; }
        = new Dictionary<string, long>();
    public IDictionary<string, long> Locations { get; set; }
        = new Dictionary<string, long>();
    protected async override Task OnParametersSetAsync() {
        if (Mode == "Edit") {
            PersonData = await Http.GetJsonAsync<Person>($"/api/people/{Id}");
        }
        Departments = (await Http.GetJsonAsync<Department[]>("/api/departments"))
            .ToDictionary(d => d.Name, d => d.Departmentid);
        Locations = (await Http.GetJsonAsync<Location[]>("/api/locations"))
            .ToDictionary(l => $"{l.City}, {l.State}", l => l.LocationId);
    }

    public string Theme => Id == 0 ? "primary" : "warning";
    public string Mode => Id == 0 ? "Create" : "Edit";

    public async Task HandleValidSubmit()  {
        await Http.SendJsonAsync(Mode == "Create" ? HttpMethod.Post : HttpMethod.Put,
            "/api/people", PersonData);
        NavManager.NavigateTo("forms");
    }
}s
```

This component uses the Blazor form features described in Chapter 36 but uses HTTP requests to read and write data to the web service created at the start of the chapter. The `GetJsonAsync<T>` method is used to read data from the web service, and the `SendJsonAsync` method is used to send either POST or PUT requests when the user submits the form.

Notice that I have not used the custom `select` component or validation components I created in Chapter 36. Sharing components between projects—especially when Blazor WebAssembly is introduced after development has started—is awkward. I expect the process to improve in future releases, but for this chapter, I have simply done without the features. As a consequence, the `select` elements do not trigger validation when a value is selected, the submit button isn't automatically disabled, and there are no restrictions on the combination of department and location.

Restart ASP.NET Core and request `http://localhost:5000/webassembly/forms`, and you will see the Blazor WebAssembly version of the form application. Click the Details button for the first item in the table, and you will see the fields for the selected object. Click the Edit button, and you will be presented with an editable form. Make a change and click the Save button, and the changes will be sent to the web service and displayed in the data table, as shown in Figure 37-5.



ID: 1
Firstname: Charles
Surname: Fuentes
Department: Development
Location: New York, NY

ID: 1
Firstname: Chuck
Surname: Fuentes
Department: Development
Location: New York, NY

ID	Name	Dept	Locati
1	Fuentes, Chuck	Development	New
2	Lara, Murphy	Sales	New
3	Hoffman, Beasley	Facilities	New
4	Lloyd, Randall	Support	San
5	Case, Guzman	Development	San
6	Jacobs, Francesca	Development	Oakl
7	Becker, Bright	Facilities	Oakl
8	Hays, Marks	Facilities	Oak
9	Trujillo, Underwood	Development	Oakl

Figure 37-5. *The completed Blazor WebAssembly form application*

Summary

In this chapter, I described Blazor WebAssembly, showed you how to add it to a project, and demonstrated how it is similar—although not identical—to the Blazor Server described in earlier chapters. In the next chapter, I explain how to use ASP.NET Core Identity to secure an application.

CHAPTER 38

Using ASP.NET Core Identity

ASP.NET Core Identity is an API from Microsoft to manage users in ASP.NET Core applications and includes support for integrating authentication and authorization into the request pipeline.

ASP.NET Core Identity is a toolkit with which you create the authorization and authentication features an application requires. There are endless integration options for features such as two-factor authentication, federation, single sign-on, and account self-service. There are options that are useful only in large corporate environments or when using cloud-hosted user management.

ASP.NET Core Identity has evolved into its own framework and is too large for me to cover in detail in this book. Instead, I have focused on the parts of the Identity API that intersect with web application development, much as I have done with Entity Framework Core. In this chapter, I show you how to add ASP.NET Core Identity to a project and explain how to consume the ASP.NET Core Identity API to create tools to perform basic user and role management. In Chapter 39, I show you how to use ASP.NET Core Identity to authenticate users and perform authorization. Table 38-1 puts ASP.NET Core Identity in context.

Table 38-1. *Putting ASP.NET Core Identity in Context*

Question	Answer
What is it?	ASP.NET Core Identity is an API for managing users.
Why is it useful?	Most applications have some features that should not be available to all users. ASP.NET Core Identity provides features to allow users to authenticate themselves and gain access to restricted features.
How is it used?	ASP.NET Core Identity is added to projects as a package and stores its data in a database using Entity Framework Core. Management of users is performed through a well-defined API, and its features are applied as attributes, as I describe in Chapter 39.
Are there any pitfalls or limitations?	ASP.NET Core Identity is complex and provides support for a wide range of authentication, authorization, and management models. It can be difficult to understand all the options, and documentation can be sparse.
Are there any alternatives?	There is no sensible alternative to ASP.NET Core Identity if a project needs to restrict access to features.

Table 38-2 summarizes the chapter.

Table 38-2. *Chapter Summary*

Problem	Solution	Listing
Preparing the application for Identity	Create the context class and use it to prepare a migration that is applied to the database	4–7
Managing user accounts	Use the `UserManager<T>` class	8–12, 15, 16
Setting a username and password policy	Use the options pattern to configure Identity	13, 14
Managing roles	Use the `RoleManager<T>` class to manage the roles and the `UserManager<T>` class to assign users to roles	17–20

Preparing for This Chapter

This chapter uses the Advanced, DataModel, and BlazorWebAssembly projects from Chapter 37. If you are using Visual Studio, open the `Advanced.sln` file you created in the previous chapter to open all three projects. If you are using Visual Studio Code, open the folder that contains the three projects.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-3`. See Chapter 1 for how to get help if you have problems running the examples.

Open a new PowerShell command prompt, navigate to the folder that contains the `Advanced.csproj` file, and run the command shown in Listing 38-1 to drop the database.

Listing 38-1. Dropping the Database

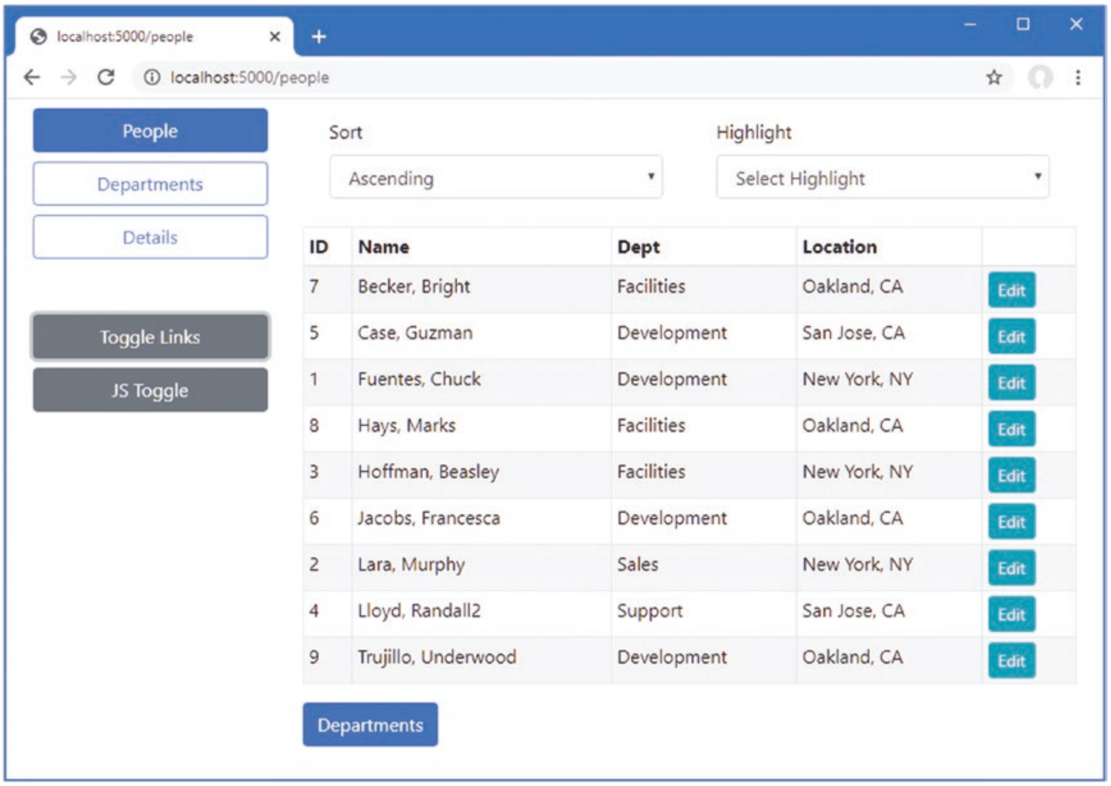
```
dotnet ef database drop --force
```

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 38-2.

Listing 38-2. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000`, which will produce the response shown in Figure 38-1.



ID	Name	Dept	Location	
7	Becker, Bright	Facilities	Oakland, CA	Edit
5	Case, Guzman	Development	San Jose, CA	Edit
1	Fuentes, Chuck	Development	New York, NY	Edit
8	Hays, Marks	Facilities	Oakland, CA	Edit
3	Hoffman, Beasley	Facilities	New York, NY	Edit
6	Jacobs, Francesca	Development	Oakland, CA	Edit
2	Lara, Murphy	Sales	New York, NY	Edit
4	Lloyd, Randall2	Support	San Jose, CA	Edit
9	Trujillo, Underwood	Development	Oakland, CA	Edit

Figure 38-1. *Running the example application*

Preparing the Project for ASP.NET Core Identity

The process for setting up ASP.NET Core Identity requires adding a package to the project, configuring the application, and preparing the database. To get started, use a PowerShell command prompt to run the command shown in Listing 38-3 in the Advanced project folder, which installs the ASP.NET Core Identity package. If you are using Visual Studio, you can install the package by selecting Project ➤ Manage NuGet Packages.

Listing 38-3. Installing ASP.NET Core Identity Packages

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 3.1.1
```

Preparing the ASP.NET Core Identity Database

ASP.NET Identity requires a database, which is managed through Entity Framework Core. To create the Entity Framework Core context class that will provide access to the Identity data, add a class file named `IdentityContext.cs` to the `Advanced/Models` folder with the code shown in Listing 38-4.

Listing 38-4. The Contents of the IdentityContext.cs File in the Models Folder of the Advanced Project

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Advanced.Models {
    public class IdentityContext: IdentityDbContext<IdentityUser> {

        public IdentityContext(DbContextOptions<IdentityContext> options)
            : base(options) { }
    }
}
```

The ASP.NET Core Identity package includes the `IdentityDbContext<T>` class, which is used to create an Entity Framework Core context class. The generic type argument `T` is used to specify the class that will represent users in the database. You can create custom user classes, but I have used the basic class, called `IdentityUser`, which provides the core Identity features.

■ **Note** Don't worry if the classes used in Listing 38-4 don't make sense. If you are unfamiliar with Entity Framework Core, then I suggest you treat the class as a black box. Changes are rarely required once the building blocks for ASP.NET Core Identity have been set up, and you can copy the files from this chapter into your own projects.

Configuring the Database Connection String

A connection string is required to tell ASP.NET Core Identity where it should store its data. In Listing 38-5, I added a connection string to the `appsettings.json` file, alongside the one used for the application data.

Listing 38-5. Adding a Connection String in the appsettings.json File in the Advanced Project

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "PeopleConnection": "Server=(localdb)\\MSSQLLocalDB;Database=People;MultipleActiveResultSets=True",
    "IdentityConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;MultipleActiveResultSets=True"
  }
}
```

The connection string specifies a LocalDB database named Identity.

■ **Note** The width of the printed page doesn't allow for sensible formatting of the connection string, which must appear in a single unbroken line. When you add the connection string to your own project, make sure that it is on a single line.

Configuring the Application

The next step is to configure ASP.NET Core so the Identity database context is set up as a service, as shown in Listing 38-6.

Listing 38-6. Configuring Identity in the Startup.cs File in the Advanced Project

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Advanced.Models;
using Microsoft.AspNetCore.ResponseCompression;
using Microsoft.AspNetCore.Identity;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddServerSideBlazor();
            services.AddSingleton<Services.ToggleService>();
            services.AddResponseCompression(opts => {
                opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
                    new[] { "application/octet-stream" });
            });

            services.AddDbContext<IdentityContext>(opts =>
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:IdentityConnection"]));
            services.AddIdentity<IdentityUser, IdentityRole>()
                .AddEntityFrameworkStores<IdentityContext>();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {

            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("controllers",
                    "controllers/{controller=Home}/{action=Index}/{id?}");
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
                endpoints.MapBlazorHub();

                endpoints.MapFallbackToClientSideBlazor<BlazorWebAssembly.Startup>
                    ("/webassembly/{*path:nonfile}", "index.html");

                endpoints.MapFallbackToPage("/_Host");
            });

            app.Map("/webassembly", opts =>
                opts.UseClientSideBlazorFiles<BlazorWebAssembly.Startup>());

            SeedData.SeedDatabase(context);
        }
    }
}
```

Creating and Applying the Identity Database Migration

The remaining step is to create the Entity Framework Core database migration and apply it to create the database. Open a new PowerShell window, navigate to the Advanced project folder, and run the commands shown in Listing 38-7.

Listing 38-7. Creating and Applying the Database Migration

```
dotnet ef migrations add --context IdentityContext Initial
dotnet ef database update --context IdentityContext
```

As I explained in earlier chapters, Entity Framework Core manages changes to database schemas through a feature called *migrations*. Now that there are two database context classes in the project, the Entity Framework Core tools require the `--context` argument to determine which context class is being used. The commands in Listing 38-7 create a migration that contains the ASP.NET Core Identity schema and apply it to the database.

RESETTING THE ASP.NET CORE IDENTITY DATABASE

If you need to reset the database, run the `dotnet ef database drop --force --context IdentityContext` command in the `Advanced` folder and then run the `dotnet ef database update --context IdentityContext` command. This will delete the existing database and create a new—and empty—replacement. Do not use these commands on production systems because you will delete user credentials. If you need to reset the main database, then run the `dotnet ef database drop --force --context DataContext` command, followed by `dotnet ef database update --context DataContext`.

Creating User Management Tools

In this section, I am going to create the tools that manage users through ASP.NET Core Identity. Users are managed through the `UserManager<T>` class, where `T` is the class chosen to represent users in the database. When I created the Entity Framework Core context class, I specified `IdentityUser` as the class to represent users in the database. This is the built-in class that is provided by ASP.NET Core Identity, and it provides the core features that are required by most applications. Table 38-3 describes the most useful `IdentityUser` properties. (There are additional properties defined by the `IdentityUser` class, but these are the ones required by most applications and are the ones I use in this book.)

SCAFFOLDING THE IDENTITY MANAGEMENT TOOLS

Microsoft provides a tool that will generate a set of Razor Pages for user management. The tool adds generic content—known as *scaffolding*—from templates to a project, which you then tailor to the application. I am not a fan of scaffolding or templates, and this is not an exception. The Microsoft Identity templates are well thought out, but they are of limited use because they focus on self-management, allowing users to create accounts, change passwords, and so on, without administrator intervention. You can adapt the templates to restrict the range of tasks that users perform, but the premise behind the features remains the same.

If you are writing the type of application where users manage their own credentials, then the scaffolding option may be worth considering and is described at `https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity`. For all other approaches, the user management API provided by ASP.NET Core Identity should be used.

Table 38-3. *Useful IdentityUser Properties*

Name	Description
`Id`	This property contains the unique ID for the user.
`UserName`	This property returns the user's username.
`Email`	This property contains the user's e-mail address.

Table 38-4 describes the `UserManagement<T>` members I use in this section to manage users.

Table 38-4. *Useful UserManager<T> Members*

Name	Description
`Users`	This property returns a sequence containing the users stored in the database.
`FindByIdAsync(id)`	This method queries the database for the user object with the specified ID.
`CreateAsync(user, password)`	This method stores a new user in the database using the specified password.
`UpdateAsync(user)`	This method modifies an existing user in the database.
`DeleteAsync(user)`	This method removes the specified user from the database.

Preparing for User Management Tools

In preparation for creating the management tools, add the expressions shown in Listing 38-8 to the `_ViewImports.cshtml` file in the Pages folder of the Advanced project.

Listing 38-8. Adding Expressions in the _ViewImports.cshtml File in the Pages Folder of the Advanced Project

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Advanced.Models
@using Microsoft.AspNetCore.Mvc.RazorPages
@using Microsoft.EntityFrameworkCore
@using System.ComponentModel.DataAnnotations
@using Microsoft.AspNetCore.Identity
@using Advanced.Pages
```

Next, create the `Pages/Users` folder in the Advanced project and add to it a Razor Layout named `_Layout.cshtml` to the `Pages/Users` folder with the content shown in Listing 38-9.

Listing 38-9. The _Layout.cshtml File in the Pages/Users Folder in the Advanced Project

```
<!DOCTYPE html>
<html>
<head>
    <title>Identity</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h5 class="bg-info text-white text-center p-2">User Administration</h5>
        @RenderBody()
    </div>
</body>
</html>
```

Add a class file named `AdminPageModel.cs` to the `Pages` folder and use it to define the class shown in Listing 38-10.

Listing 38-10. The AdminPageModel.cs File in the Pages Folder in the Advanced Project

```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Advanced.Pages {
    public class AdminPageModel: PageModel {

    }
}
```

This class will be the base for the page model classes defined in this section. As you will see in Chapter 39, a common base class is useful when it comes to securing the application.

Enumerating User Accounts

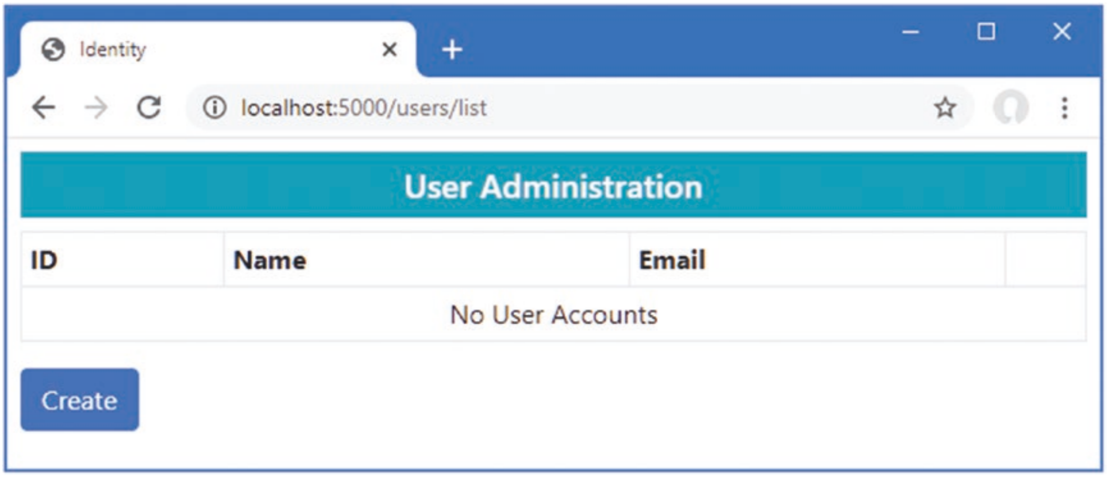
Although the database is currently empty, I am going to start by creating a Razor Page that will enumerate user accounts. Add a Razor Page named `List.cshtml` to the `Pages/Users` folder in the Advanced project with the content shown in Listing 38-11.

Listing 38-11. The Contents of the List.cshtml File in the Pages/Users Folder in the Advanced Project

```
@page
@model ListModel

<table class="table table-sm table-bordered">
    <tr><th>ID</th><th>Name</th><th>Email</th><th></th></tr>
    @if (Model.Users.Count() == 0) {
        <tr><td colspan="4" class="text-center">No User Accounts</td></tr>
    } else {
        foreach (IdentityUser user in Model.Users) {
            <tr>
                <td>@user.Id</td>
                <td>@user.UserName</td>
                <td>@user.Email</td>
                <td class="text-center">
                    <form asp-page="List" method="post">
                        <input type="hidden" name="Id" value="@user.Id" />
                        <a class="btn btn-sm btn-warning" asp-page="Editor"
                            asp-route-id="@user.Id" asp-route-mode="edit">Edit</a>
                        <button type="submit" class="btn btn-sm btn-danger">
                            Delete
                        </button>
                    </form>
                </td>
            </tr>
        }
    }
</table>

<a class="btn btn-primary" asp-page="create">Create</a>

@functions {

    public class ListModel : AdminPageModel {
        public UserManager<IdentityUser> UserManager;

        public ListModel(UserManager<IdentityUser> userManager) {
            UserManager = userManager;
        }

        public IEnumerable<IdentityUser> Users { get; set; }

        public void OnGet() {
            Users = UserManager.Users;
        }
    }
}
```

The `UserManager<IdentityUser>` class is set up as a service so that it can be consumed via dependency injection. The `Users` property returns a collection of `IdentityUser` objects, which can be used to enumerate the user accounts. This Razor Page displays the users in a table, with buttons that allow each user to be edited or deleted, although this won't be visible initially because a placeholder message is shown when there are no user objects to display. There is a button that navigates to a Razor Page named `Create`, which I define in the next section.

Restart ASP.NET and request `http://localhost:5000/users/list` to see the (currently empty) data table, which is shown in Figure 38-2.



User Administration

ID	Name	Email	
No User Accounts			

Figure 38-2. Enumerating users

Creating Users

Add a Razor Page named `Create.cshtml` to the `Pages/Users` folder with the content shown in Listing 38-12.

Listing 38-12. The Contents of the Create.cshtml File in the Pages/Users Folder of the Advanced Project

```
@page
@model CreateModel

<h5 class="bg-primary text-white text-center p-2">Create User</h5>
<form method="post">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label>User Name</label>
        <input name="UserName" class="form-control" value="@Model.UserName" />
    </div>
    <div class="form-group">
        <label>Email</label>
        <input name="Email" class="form-control" value="@Model.Email" />
    </div>
    <div class="form-group">
        <label>Password</label>
        <input name="Password" class="form-control" value="@Model.Password" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
    <a class="btn btn-secondary" asp-page="list">Back</a>
</form>

@functions {

    public class CreateModel : AdminPageModel {
        public UserManager<IdentityUser> UserManager;

        public CreateModel(UserManager<IdentityUser> usrManager) {
            UserManager = usrManager;
        }
        [BindProperty][Required]
        public string UserName { get; set; }

        [BindProperty][Required][EmailAddress]
        public string Email { get; set; }

        [BindProperty][Required]
        public string Password { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user =
                    new IdentityUser { UserName = UserName, Email = Email };
                IdentityResult result =
                    await UserManager.CreateAsync(user, Password);
                if (result.Succeeded) {
                    return RedirectToPage("List");
                }
                foreach (IdentityError err in result.Errors) {
                    ModelState.AddModelError("", err.Description);
                }
            }
            return Page();
        }
    }
}
```

Even though ASP.NET Core Identity data is stored using Entity Framework Core, you don't work directly with the database context class. Instead, data is managed through the methods provided by the `UserManager<T>` class. New users are created using the `CreateAsync` method, which accepts an `IdentityUser` object and a password string as arguments.

This Razor Page defines three properties that are subject to model binding. The `UserName` and `Email` properties are used to configure the `IdentityUser` object, which is combined with the value bound to the `Password` property to call the `CreateAsync` method. These properties are configured with validation attributes, which ensure that values are supplied and that the `Email` property is a formatted e-mail address.

The result of the `CreateAsync` method is a `Task<IdentityResult>` object, which indicates the outcome of the create operation, using the properties described in Table 38-5.

Table 38-5. *The Properties Defined by the IdentityResult Class*

Name	Description
`Succeeded`	Returns `true` if the operation succeeded.
`Errors`	Returns a sequence of `IdentityError` objects that describe the errors encountered while attempting the operation. Each `IdentityError` object provides a `Description` property that summarizes the problem.

I inspect the `Succeeded` property to determine whether a new user has been created in the database. If the `Succeeded` property is `true`, then the client is redirected to the `List` page so that the list of users is displayed, reflecting the new addition.

```
...
if (result.Succeeded) {
    return RedirectToPage("List");
}
foreach (IdentityError err in result.Errors) {
    ModelState.AddModelError("", err.Description);
}
...
```

If the Succeeded property is false, then the sequence of IdentityError objects provided by the Errors property is enumerated, with the Description property used to create a model-level validation error using the ModelState.AddModelError method.

To test the ability to create a new user account, restart ASP.NET Core and request http://localhost:5000/users/list. Click the Create button and fill in the form with the values shown in Table 38-6.

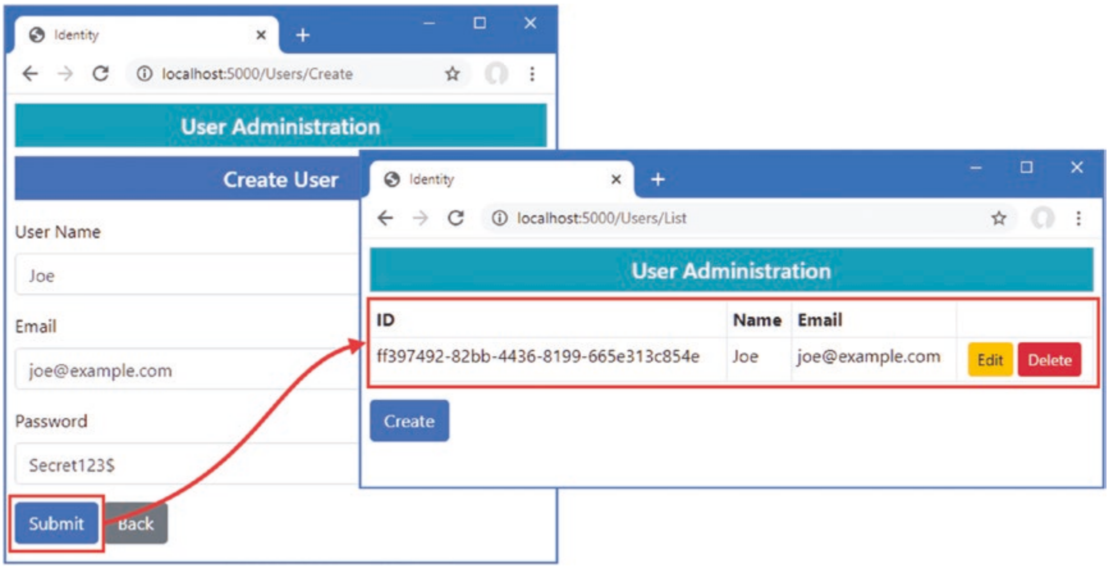
■ **Tip** There are domains reserved for testing, including example.com. You can see a complete list at https://tools.ietf.org/html/rfc2606.

Table 38-6. *The Values for Creating an Example User*

Field	*Description*
Name	Joe
Email	***joe@example.com***
Password	Secret123$

Once you have entered the values, click the Submit button. ASP.NET Core Identity will create the user in the database, and the browser will be redirected, as shown in Figure 38-3. (You will see a different ID value because IDs are randomly generated for each user.)

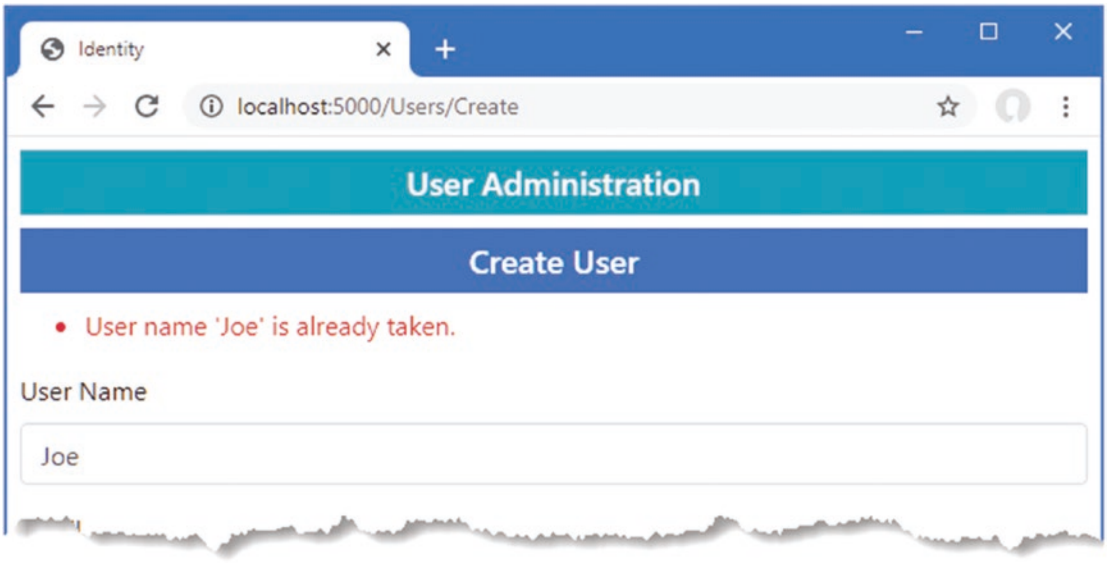
■ **Note** I used a regular input element for the Password field to make it easier to follow the examples in this chapter. For real projects, it is a good idea to set the input element's type attribute to password so that the characters entered cannot be seen.



ID	Name	Email	
ff397492-82bb-4436-8199-665e313c854e	Joe	joe@example.com	Edit Delete

Figure 38-3. *Creating a new user*

Click the Create button again and enter the same details into the form, using the values in Table 38-6. This time you will see an error reported through the model validation summary when you click the Create button, as shown in Figure 38-4. This is an example of an error returned through the IdentityResult object produced by the CreateAsync method.



- User name 'Joe' is already taken.

User Name

Joe

Figure 38-4. *An error when creating a new user*

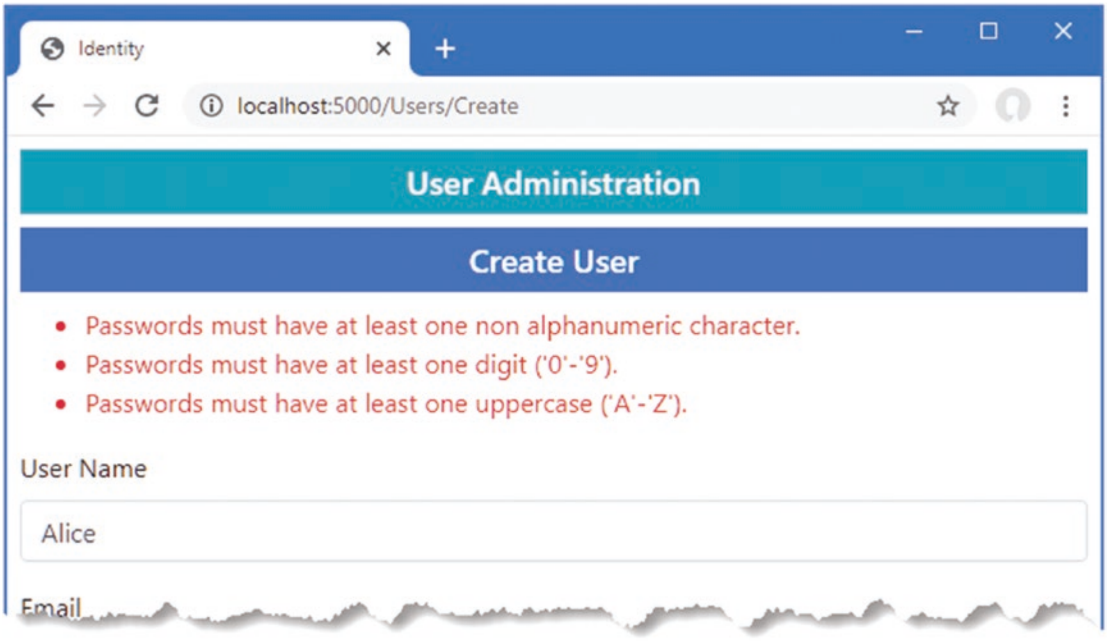
Validating Passwords

One of the most common requirements, especially for corporate applications, is to enforce a password policy. You can see the default policy by navigating to `http://localhost:5000/Users/Create` and filling out the form with the data shown in Table 38-7.

Table 38-7. *The Values for Creating an Example User*

Field	Description
`Name`	`Alice`
`Email`	alice@example.com
`Password`	`secret`

When you submit the form, ASP.NET Core Identity checks the candidate password and generates errors if it doesn't match the password, as shown in Figure 38-5.



User Administration

Create User

- Passwords must have at least one non alphanumeric character.
- Passwords must have at least one digit ('0'-'9').
- Passwords must have at least one uppercase ('A'-'Z').

User Name

Alice

Email

Figure 38-5. *Password validation errors*

The password validation rules are configured using the options pattern, as shown in Listing 38-13.

Listing 38-13. Configuring Password Validation in the Startup.cs File in the Advanced Project

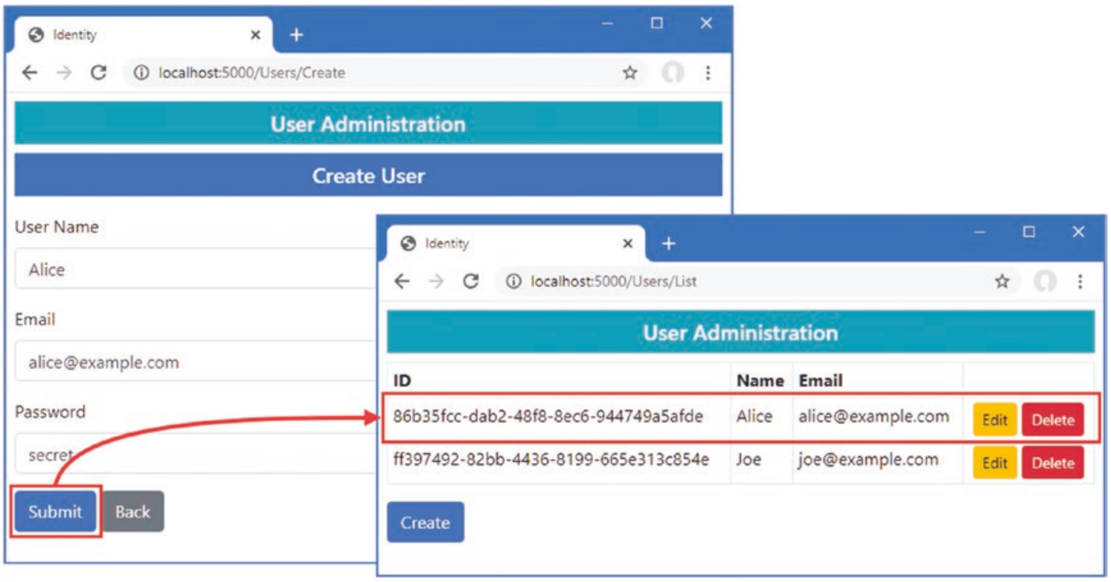
```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:PeopleConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddServerSideBlazor();
    services.AddSingleton<Services.ToggleService>();

    services.AddResponseCompression(opts => {
        opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "application/octet-stream" });
    });

    services.AddDbContext<IdentityContext>(opts =>
        opts.UseSqlServer(Configuration["ConnectionStrings:IdentityConnection"]));
    services.AddIdentity<IdentityUser, IdentityRole>()
        .AddEntityFrameworkStores<IdentityContext>();

    services.Configure<IdentityOptions>(opts => {
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
    });
}
...
```

ASP.NET Core Identity is configured using the `IdentityOptions` class, whose `Password` property returns a `PasswordOptions` class that configures password validation using the properties described in Table 38-8.

Table 38-8. *The PasswordOptions Properties*

Name	Description
`RequiredLength`	This `int` property is used to specify the minimum length for passwords.
`RequireNonAlphanumeric`	Setting this `bool` property to `true` requires passwords to contain at least one character that is not a letter or a digit.
`RequireLowercase`	Setting this `bool` property to `true` requires passwords to contain at least one lowercase character.
`RequireUppercase`	Setting this `bool` property to `true` requires passwords to contain at least one uppercase character.
`RequireDigit`	Setting this `bool` property to `true` requires passwords to contain at least one numeric character.

In the listing, I specified that passwords must have a minimum length of six characters and disabled the other constraints. This isn't something that you should do without careful consideration in a real project, but it allows for an effective demonstration. Restart ASP.NET Core, request `http://localhost:5000/users/create`, and fill out the form using the details from Table 38-7. When you click the Submit button, the password will be accepted by the new validation rules, and a new user will be created, as shown in Figure 38-6.



ID	Name	Email	
86b35fcc-dab2-48f8-8ec6-944749a5afde	Alice	alice@example.com	Edit Delete
ff397492-82bb-4436-8199-665e313c854e	Joe	joe@example.com	Edit Delete

Figure 38-6. *Changing the password validation rules*

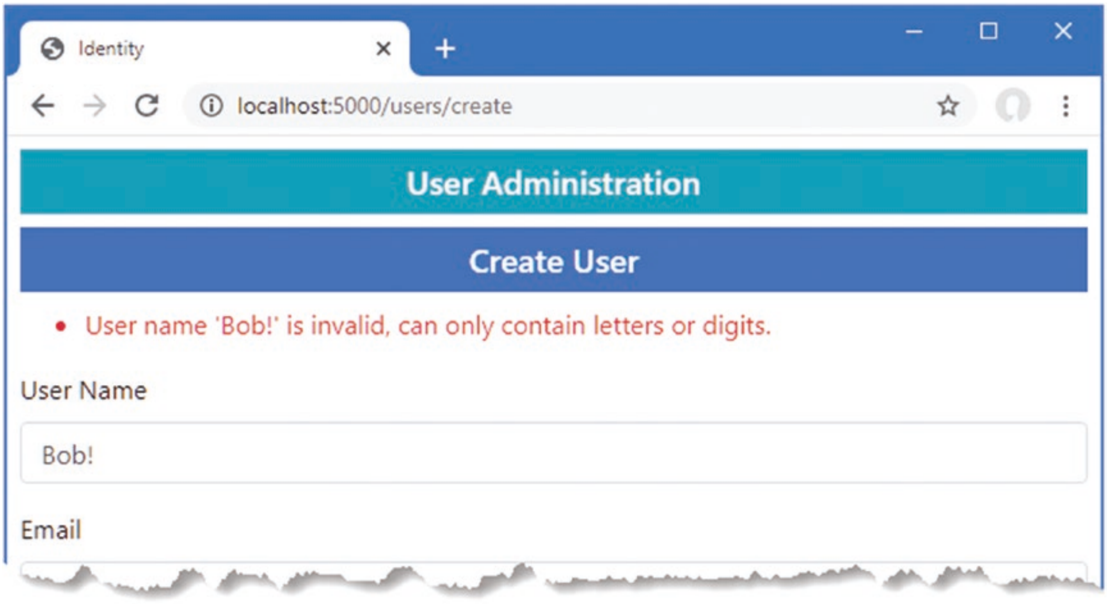
Validating User Details

Validation is also performed on usernames and e-mail addresses when accounts are created. To see how validation is applied, request `http://localhost:5000/users/create` and fill out the form using the values shown in Table 38-9.

Table 38-9. The Values for Creating an Example User

Field	Description
`Name`	`Bob!`
`Email`	alice@example.com
`Password`	`secret`

Click the Submit button, and you will see the error message shown in Figure 38-7.



- User name 'Bob!' is invalid, can only contain letters or digits.

User Name: Bob!

Email

Figure 38-7. A user details validation error

Validation can be configured with the options pattern, using the `User` property defined by the `IdentityOptions` class. This class returns a `UserOptions` class, whose properties are described in Table 38-10.

Table 38-10. The UserOptions Properties

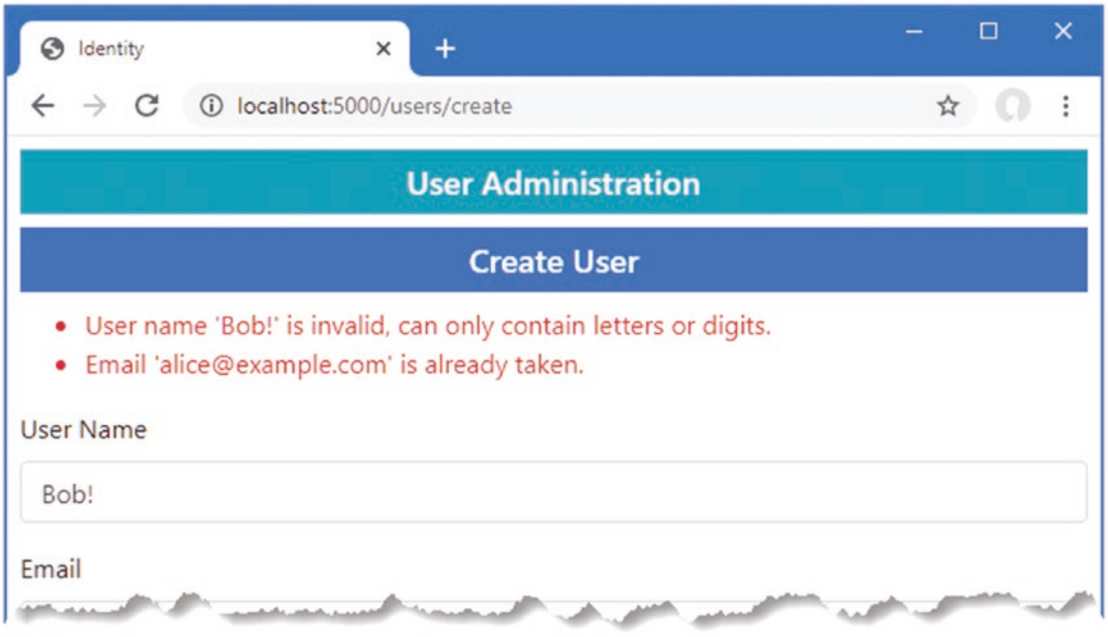
Name	Description
`AllowedUserNameCharacters`	This `string` property contains all the legal characters that can be used in a username. The default value specifies a–z, A–Z, and 0–9 and the hyphen, period, underscore, and `@` characters. This property is not a regular expression, and every legal character must be specified explicitly in the string.
`RequireUniqueEmail`	Setting this `bool` property to `true` requires new accounts to specify e-mail addresses that have not been used previously.

In Listing 38-14, I have changed the configuration of the application so that unique e-mail addresses are required and so that only lowercase alphabetic characters are allowed in usernames.

Listing 38-14. Changing the User Validation Settings in the Startup.cs File in the Advanced Project

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:PeopleConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllersWithViews().AddRazorRuntimeCompilation();
    services.AddRazorPages().AddRazorRuntimeCompilation();
    services.AddServerSideBlazor();
    services.AddSingleton<Services.ToggleService>();

    services.AddResponseCompression(opts => {
        opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "application/octet-stream" });
    });

    services.AddDbContext<IdentityContext>(opts =>
        opts.UseSqlServer(Configuration["ConnectionStrings:IdentityConnection"]));
    services.AddIdentity<IdentityUser, IdentityRole>()
        .AddEntityFrameworkStores<IdentityContext>();

    services.Configure<IdentityOptions>(opts => {
        opts.Password.RequiredLength = 6;
        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
        opts.User.RequireUniqueEmail = true;
        opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
    });
}
...
```

Restart ASP.NET Core, request `http://localhost:5000/users/create`, and fill out the form with the values in Table 38-9. Click the Submit button, and you will see that the e-mail address now causes an error. The username still contains illegal characters and is also flagged as an error, as shown in Figure 38-8.



- User name 'Bob!' is invalid, can only contain letters or digits.
- Email 'alice@example.com' is already taken.

User Name: Bob!

Email

Figure 38-8. *Validating user detail*

Editing Users

To add support for editing users, add a Razor Page named `Editor.cshtml` to the `Pages/Users` folder of the Advanced project with the content shown in Listing 38-15.

Listing 38-15. The Contents of the Editor.cshtml File in the Pages/Users Folder of the Advanced Project

```
@page "{id}"
@model EditorModel

<h5 class="bg-warning text-white text-center p-2">Edit User</h5>
<form method="post">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label>ID</label>
        <input name="Id" class="form-control" value="@Model.Id" disabled />
        <input name="Id" type="hidden" value="@Model.Id" />
    </div>
    <div class="form-group">
        <label>User Name</label>
        <input name="UserName" class="form-control" value="@Model.UserName" />
    </div>
    <div class="form-group">
        <label>Email</label>
        <input name="Email" class="form-control" value="@Model.Email" />
    </div>
    <div class="form-group">
        <label>New Password</label>
        <input name="Password" class="form-control" value="@Model.Password" />
    </div>
    <button type="submit" class="btn btn-warning">Submit</button>
    <a class="btn btn-secondary" asp-page="list">Back</a>
</form>
```

```
@functions {

    public class EditorModel : AdminPageModel {
        public UserManager<IdentityUser> UserManager;

        public EditorModel(UserManager<IdentityUser> usrManager) {
            UserManager = usrManager;
        }

        [BindProperty][Required]
        public string Id { get; set; }

        [BindProperty][Required]
        public string UserName { get; set; }

        [BindProperty][Required][EmailAddress]
        public string Email { get; set; }

        [BindProperty]
        public string Password { get; set; }

        public async Task OnGetAsync(string id) {
            IdentityUser user = await UserManager.FindByIdAsync(id);
            Id = user.Id; UserName = user.UserName; Email = user.Email;
        }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user = await UserManager.FindByIdAsync(Id);
                user.UserName = UserName;
                user.Email = Email;
                IdentityResult result = await UserManager.UpdateAsync(user);
                if (result.Succeeded && !String.IsNullOrEmpty(Password)) {
                    await UserManager.RemovePasswordAsync(user);
                    result = await UserManager.AddPasswordAsync(user, Password);
                }
                if (result.Succeeded) {
                    return RedirectToPage("List");
                }
                foreach (IdentityError err in result.Errors) {
                    ModelState.AddModelError("", err.Description);
                }

            }
            return Page();
        }
    }
}
```

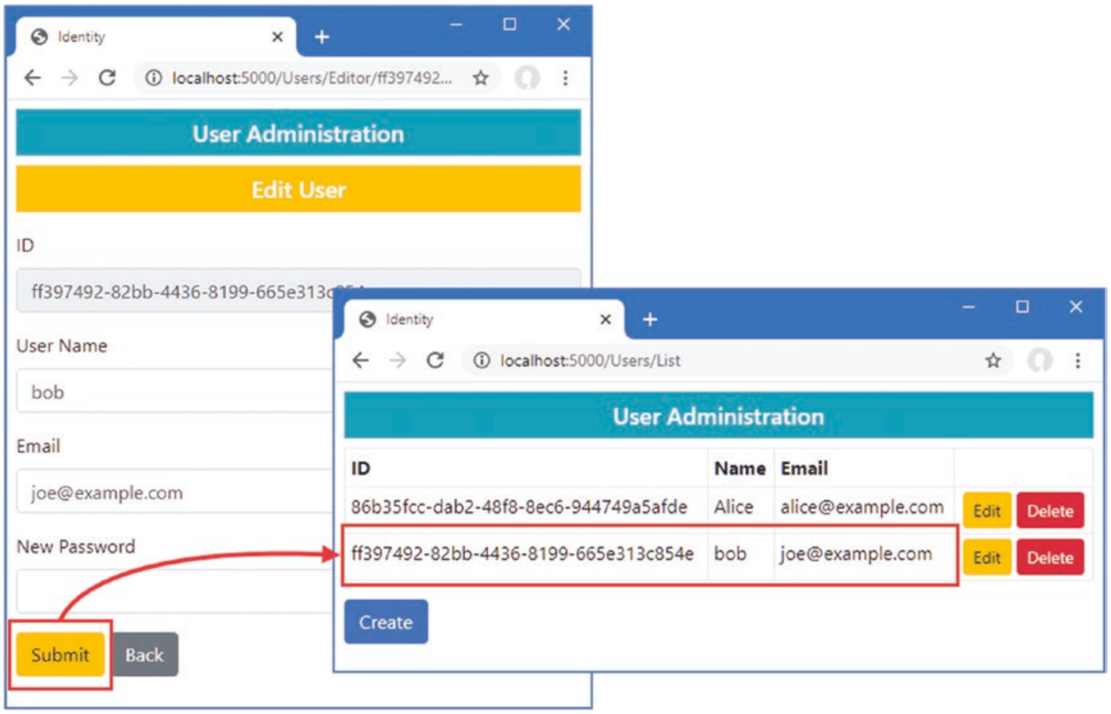
The `Editor` page uses the `UserManager<T>.FindByIdAsync` method to locate the user, querying the database with the `id` value received through the routing system and received as an argument to the `OnGetAsync` method. The values from the `IdentityUser` object returned by the query are used to populate the properties that are displayed by the view part of the page, ensuring that the values are not lost if the page is redisplayed due to validation errors.

When the user submits the form, the FindByIdAsync method is used to query the database for the IdentityUser object, which is updated with the UserName and Email values provided in the form. Passwords required a different approach and must be removed from the user object before a new password is assigned, like this:

```
...
await UserManager.RemovePasswordAsync(user);
result = await UserManager.AddPasswordAsync(user, Password);
...
```

The Editor page changes the password only if the form contains a Password value and if the updates for the UserName and Email fields have been successful. Errors from ASP.NET Core Identity are presented as validation messages, and the browser is redirected to the List page after a successful update. Request http://localhost:5000/Users/List, click the Edit button for Joe, and change the UserName field to **bob**, with all lowercase characters. Click the Submit button, and you will see the change reflected in the list of users, as shown in Figure 38-9.

■ **Note** You will see an error if you click the Edit button for the Alice account and click Submit without making changes. This is because the account was created before the validation policy was changed. ASP.NET Core Identity applies validation checks for updates, leading to the odd situation where the data in the database can be read—and used—but must be changed for the user to be updated.



ID	Name	Email	
86b35fcc-dab2-48f8-8ec6-944749a5afde	Alice	alice@example.com	Edit Delete
ff397492-82bb-4436-8199-665e313c854e	bob	joe@example.com	Edit Delete

Figure 38-9. *Editing a user*

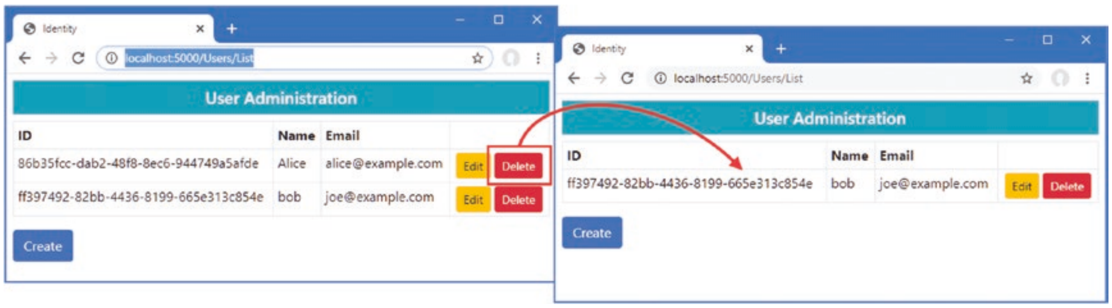
Deleting Users

The last feature I need for my basic user management application is the ability to delete users, as shown in Listing 38-16.

Listing 38-16. Deleting Users in the List.cshtml File in the Pages/Users Folder in the Advanced Project

```
...
@functions {

    public class ListModel : AdminPageModel {
        public UserManager<IdentityUser> UserManager;

        public ListModel(UserManager<IdentityUser> userManager) {
            UserManager = userManager;
        }

        public IEnumerable<IdentityUser> Users { get; set; }

        public void OnGet() {
            Users = UserManager.Users;
        }

        public async Task<IActionResult> OnPostAsync(string id) {
            IdentityUser user = await UserManager.FindByIdAsync(id);
            if (user != null) {
                await UserManager.DeleteAsync(user);
            }
            return RedirectToPage();
        }
    }
}
...
```

The `List` page already displays a Delete button for each user in the data table, which submits a POST request containing the `Id` value for the `IdentityUser` object to be removed. The `OnPostAsync` method receives the `Id` value and uses it to query Identity using the `FindByIdAsync` method, passing the object that is returned to the `DeleteAsync` method, which deletes it from the database. To check the delete functionality, request `http://localhost:5000/Users/List` and click Delete for the `Alice` account. The user object will be removed, as shown in Figure 38-10.



ID	Name	Email
86b35fcc-dab2-48f8-8ec6-944749a5afde	Alice	alice@example.com
ff397492-82bb-4436-8199-665e313c854e	bob	joe@example.com

ID	Name	Email
ff397492-82bb-4436-8199-665e313c854e	bob	joe@example.com

Figure 38-10. *Deleting a user*

Creating Role Management Tools

Some applications enforce only two levels of authorization: authenticated users are allowed access to all the application's features, while unauthenticated users have less—or no—access. The SportsStore application in Part 1 followed this approach: there was one user, and once authenticated, they had access to all the application's features, including administration tools, while unauthenticated users were restricted to the public store features.

ASP.NET Core Identity supports *roles* for applications that require more granular authorization. Users are assigned to one or more roles, and their membership of those roles determines which features are accessible. In the sections that follow, I show you how to build tools to create and manage roles.

Roles are managed through the `RoleManager<T>` class, where `T` is the representation of roles in the database. When I configured ASP.NET Core Identity at the start of the chapter, I selected `IdentityRole`, which is the built-in class that Identity provides to describe a role, which means that I will be using the `RoleManager<IdentityRole>` class in these examples. The `RoleManager<T>` class defines the methods and properties shown in Table 38-11 that allow roles to be created and managed.

Table 38-11. *The Members Defined by the RoleManager<T> Class*

Name	Description
`CreateAsync(role)`	Creates a new role
`DeleteAsync(role)`	Deletes the specified role
`FindByIdAsync(id)`	Finds a role by its ID
`FindByNameAsync(name)`	Finds a role by its name
`RoleExistsAsync(name)`	Returns `true` if a role with the specified name exists
`UpdateAsync(role)`	Stores changes to the specified role
`Roles`	Returns an enumeration of the roles that have been defined

Table 38-12 describes the key properties defined by the `IdentityRole` class.

Table 38-12. *Useful IdentityRole Properties*

Name	Description
`Id`	This property contains the unique ID for the role.
`Name`	This property returns the role name.

Although roles are managed through the `RoleManager<T>` class, membership of roles is managed through the methods provided by `UserManager<T>` described in Table 38-13.

Table 38-13. *The UserManager<T> Methods for Managing Role Membership*

Name	Description
`AddToRoleAsync(user, role)`	This method adds a user to a role.
`RemoveFromRoleAsync(user, role)`	This method removes a user from a role.
`GetRolesAsync(user)`	This method returns the roles for which the user is a member.
`GetUsersInRoleAsync(role)`	This method returns users who are members of the specified role.
`IsInRoleAsync(user, role)`	This method returns `true` if the user is a member of the specified role.

Preparing for Role Management Tools

To prepare for the role management tools, create the `Pages/Roles` folder in the Advanced project and add to it a Razor layout named `_Layout.cshtml` with the content shown in Listing 38-17.

Listing 38-17. The Contents of the _Layout.cshtml File in the Pages/Roles Folder in the Advanced Project

```
<!DOCTYPE html>
<html>
<head>
    <title>Identity</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        <h5 class="bg-secondary text-white text-center p-2">Role Administration</h5>
        @RenderBody()
    </div>
</body>
</html>
```

This layout will ensure there is an obvious difference between the user and role management tools.

Enumerating and Deleting Roles

Add a Razor Page named `List.cshtml` to the `Pages/Roles` folder in the Advanced project with the content shown in Listing 38-18.

Listing 38-18. The Contents of the List.cshtml File in the Pages/Roles Folder in the Advanced Project

```
@page
@model ListModel

<table class="table table-sm table-bordered">
    <tr><th>ID</th><th>Name</th><th>Members</th><th></th></tr>
    @if (Model.Roles.Count() == 0) {
        <tr><td colspan="4" class="text-center">No Roles</td></tr>
    } else {
        foreach (IdentityRole role in Model.Roles) {
            <tr>
                <td>@role.Id</td>
                <td>@role.Name</td>
                <td>@(await Model.GetMembersString(role.Name))</td>
                <td class="text-center">
                    <form asp-page="List" method="post">
                        <input type="hidden" name="Id" value="@role.Id" />
                        <a class="btn btn-sm btn-warning" asp-page="Editor"
                            asp-route-id="@role.Id" asp-route-mode="edit">Edit</a>
                        <button type="submit" class="btn btn-sm btn-danger">
                            Delete
                        </button>
                    </form>
                </td>
            </tr>
        }
    }
</table>
<a class="btn btn-primary" asp-page="create">Create</a>

@functions {

    public class ListModel : AdminPageModel {
        public UserManager<IdentityUser> UserManager;
        public RoleManager<IdentityRole> RoleManager;
        public ListModel(UserManager<IdentityUser> userManager,
                RoleManager<IdentityRole> roleManager) {
            UserManager = userManager;
            RoleManager = roleManager;
        }

        public IEnumerable<IdentityRole> Roles { get; set; }

        public void OnGet() {
            Roles = RoleManager.Roles;
        }

        public async Task<string> GetMembersString(string role) {
            IEnumerable<IdentityUser> users
                = (await UserManager.GetUsersInRoleAsync(role));
            string result = users.Count() == 0
                ? "No members"
                : string.Join(", ", users.Take(3).Select(u => u.UserName).ToArray());
            return users.Count() > 3 ? $"{result}, (plus others)" : result;
        }

        public async Task<IActionResult> OnPostAsync(string id) {
            IdentityRole role = await RoleManager.FindByIdAsync(id);
            await RoleManager.DeleteAsync(role);
            return RedirectToPage();
        }
    }
}
```

The roles are enumerated, along with the names of up to three of the role members or a placeholder message if there are no members. There is also a Create button, and each role is presented with Edit and Delete buttons, following the same pattern I used for the user management tools.

The Delete button sends a POST request back to the Razor Page. The `OnPostAsync` method uses the `FindByIdAsync` method to retrieve the role object, which is passed to the `DeleteAsync` method to remove it from the database.

Creating Roles

Add a Razor Page named `Create.cshtml` in the `Pages/Roles` folder in the Advanced project with the contents shown in Listing 38-19.

Listing 38-19. The Contents of the Create.cshtml File in the Pages/Roles Folder in the Advanced Project

```
@page
@model CreateModel

<h5 class="bg-primary text-white text-center p-2">Create Role</h5>
<form method="post">
    <div asp-validation-summary="All" class="text-danger"></div>
    <div class="form-group">
        <label>Role Name</label>
        <input name="Name" class="form-control" value="@Model.Name" />
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
    <a class="btn btn-secondary" asp-page="list">Back</a>
</form>
```

```
@functions {

    public class CreateModel : AdminPageModel {
        public RoleManager<IdentityRole> RoleManager;

        public CreateModel(UserManager<IdentityUser> userManager,
                RoleManager<IdentityRole> roleManager) {
            RoleManager = roleManager;
        }

        [BindProperty][Required]
        public string Name { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityRole role = new IdentityRole { Name = Name };
                IdentityResult result = await RoleManager.CreateAsync(role);
                if (result.Succeeded) {
                    return RedirectToPage("List");
                }
                foreach (IdentityError err in result.Errors) {
                    ModelState.AddModelError("", err.Description);
                }
            }
            return Page();
        }
    }
}
```

The user is presented with a form containing an `input` element to specify the name of the new role. When the form is submitted, the `OnPostAsync` method creates a new `IdentityRole` object and passes it to the `CreateAsync` method.

Assigning Role Membership

To add support for managing role memberships, add a Razor Page named `Editor.cshtml` to the `Pages/Roles` folder in the Advanced project, with the content shown in Listing 38-20.

Listing 38-20. The Contents of the Editor.cshtml File in the Pages/Roles Folder in the Advanced Project

```
@page "{id}"
@model EditorModel

<h5 class="bg-primary text-white text-center p-2">Edit Role: @Model.Role.Name</h5>

<form method="post">
    <input type="hidden" name="rolename" value="@Model.Role.Name" />
    <div asp-validation-summary="All" class="text-danger"></div>
    <h5 class="bg-secondary text-white p-2">Members</h5>
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>User</th><th>Email</th><th></th></tr></thead>
        <tbody>
            @if ((await Model.Members()).Count() == 0) {
                <tr><td colspan="3" class="text-center">No members</td></tr>
            }
            @foreach (IdentityUser user in await Model.Members()) {
                <tr>
                    <td>@user.UserName</td>
                    <td>@user.Email</td>
                    <td>
                        <button asp-route-userid="@user.Id"
                                class="btn btn-primary btn-sm" type="submit">
                            Change
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>

    <h5 class="bg-secondary text-white p-2">Non-Members</h5>
    <table class="table table-sm table-striped table-bordered">
        <thead><tr><th>User</th><th>Email</th><th></th></tr></thead>
        <tbody>
            @if ((await Model.NonMembers()).Count() == 0) {
                <tr><td colspan="3" class="text-center">No non-members</td></tr>
            }
            @foreach (IdentityUser user in await Model.NonMembers()) {
                <tr>
                    <td>@user.UserName</td>
                    <td>@user.Email</td>
                    <td>
                        <button asp-route-userid="@user.Id"
                            class="btn btn-primary btn-sm" type="submit">
                                Change
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</form>

<a class="btn btn-secondary" asp-page="list">Back</a>

@functions {

    public class EditorModel : AdminPageModel {
        public UserManager<IdentityUser> UserManager;
        public RoleManager<IdentityRole> RoleManager;

        public EditorModel(UserManager<IdentityUser> userManager,
                RoleManager<IdentityRole> roleManager) {
            UserManager = userManager;
            RoleManager = roleManager;
        }

        public IdentityRole Role { get; set; }

        public Task<IList<IdentityUser>> Members() =>
                UserManager.GetUsersInRoleAsync(Role.Name);
        public async Task<IEnumerable<IdentityUser>> NonMembers() =>
                UserManager.Users.ToList().Except(await Members());

        public async Task OnGetAsync(string id) {
            Role = await RoleManager.FindByIdAsync(id);
        }

        public async Task<IActionResult> OnPostAsync(string userid,
                string rolename) {
            Role = await RoleManager.FindByNameAsync(rolename);
            IdentityUser user = await UserManager.FindByIdAsync(userid);
            IdentityResult result;
            if (await UserManager.IsInRoleAsync(user, rolename)) {
                result = await UserManager.RemoveFromRoleAsync(user, rolename);
            } else {
                result = await UserManager.AddToRoleAsync(user, rolename);
            }
            if (result.Succeeded) {
                return RedirectToPage();
            } else {
                foreach (IdentityError err in result.Errors) {
                    ModelState.AddModelError("", err.Description);
                }
                return Page();
            }
        }
    }
}
```

The user is presented with a table showing the users who are members of the role and with a table showing nonmembers. Each row contains a Change button that submits the form. The `OnPostAsync` method uses the `UserManager.FindByIdAsync` method to retrieve the user object from the database. The `IsInRoleAsync` method is used to determine whether the user is a member of the role, and the `AddToRoleAsync` and `RemoveFromRoleAsync` methods are used to add and remove the user, respectively.

Restart ASP.NET Core and request `http://localhost:5000/roles/list`. The list will be empty because there are no roles in the database. Click the Create button, enter **Admins** into the text field, and click the Submit button to create a new role. Once the role has been created, click the Edit button, and you will see the list of users who can be added to the role. Clicking the Change button will move the user in and out of the role. Click back, and the list will be updated to show the users who are members of the role, as shown in Figure 38-11.

■ **Caution** ASP.NET Core Identity revalidates user details when changing role assignments, which will result in an error if you try to modify a user whose details do not match the current restrictions, which happens when restrictions are introduced after the application has been deployed and the database is already populated with users created under the old roles. It is for this reason that the Razor Page in Listing 38-20 checks the result from the operations to add or remove users from a role and displays any errors as validation messages.

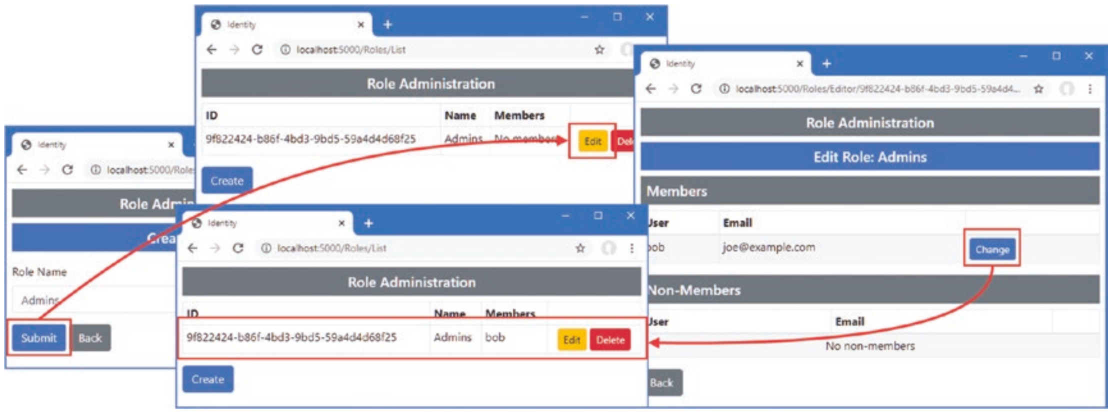


Figure 38-11. *Managing roles*

Summary

In this chapter, I showed you how to add ASP.NET Core Identity to a project and prepare its database to store users and roles. I described the basic ASP.NET Core Identity API and showed you how it can be used to create tools to manage users and roles. In the next chapter, I show you how to apply ASP.NET Core Identity to control access to controllers, Razor Pages, Blazor applications, and web services.

CHAPTER 39

Applying ASP.NET Core Identity

In this chapter, I explain how ASP.NET Core Identity is applied to authenticate users and authorize access to application features. I create the features required for users to establish their identity, explain how access to endpoints can be controlled, and demonstrate the security features that Blazor provides. I also show two different ways to authenticate web service clients. Table 39-1 summarizes the chapter.

Table 39-1. *Chapter Summary*

Problem	Solution	Listing
Authenticating users	Use the `SignInManager<T>` class to validate the credentials users provide and use the built-in middleware to trigger authentication	3–18
Restricting access to endpoints	Use the `Authorize` attribute and the built-in middleware to control access	9–13
Restricting access to Blazor components	Use the `Authorize` attribute and the built-in Razor Components to control access	14–17
Restricting access to web services	Use cookie authentication or bearer tokens	18–30

Preparing for This Chapter

This chapter uses the projects from Chapter 38. To prepare for this chapter, I am going to reset both the application data and ASP.NET Core Identity databases and create new users and roles. Open a new command prompt and run the commands shown in Listing 39-1 in the `Advanced` project folder, which contains the `Advanced.csproj` file. These commands remove the existing databases and re-create them.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/apress/pro-asp.net-core-3`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 39-1. Re-creating the Project Databases

```
dotnet ef database drop --force --context DataContext
dotnet ef database drop --force --context IdentityContext
dotnet ef database update --context DataContext
dotnet ef database update --context IdentityContext
```

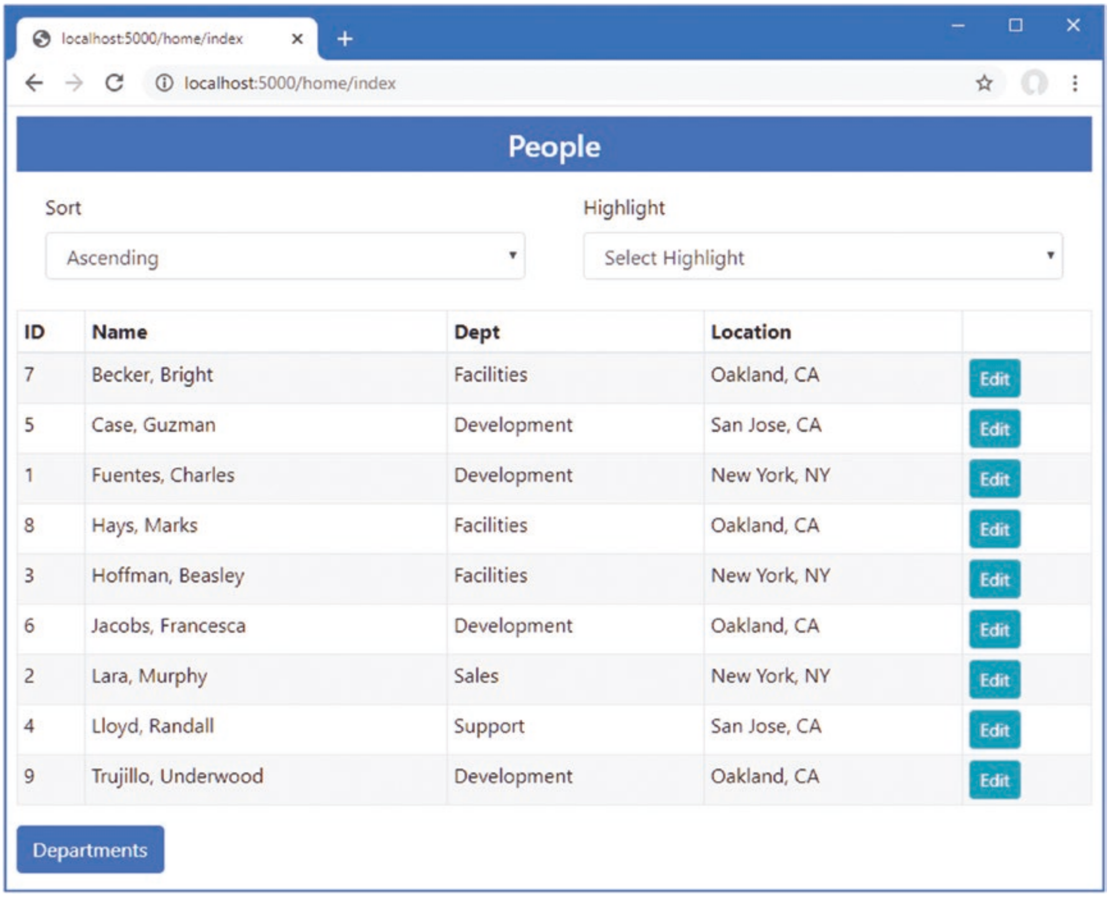
Now that the application contains multiple database context classes, the Entity Framework Core commands require the `--context` argument to select the context that a command applies to.

Select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 39-2.

Listing 39-2. Running the Example Application

```
dotnet run
```

Use a browser to request `http://localhost:5000/home/index`, which will produce the response shown in Figure 39-1.



ID	Name	Dept	Location	
7	Becker, Bright	Facilities	Oakland, CA	Edit
5	Case, Guzman	Development	San Jose, CA	Edit
1	Fuentes, Charles	Development	New York, NY	Edit
8	Hays, Marks	Facilities	Oakland, CA	Edit
3	Hoffman, Beasley	Facilities	New York, NY	Edit
6	Jacobs, Francesca	Development	Oakland, CA	Edit
2	Lara, Murphy	Sales	New York, NY	Edit
4	Lloyd, Randall	Support	San Jose, CA	Edit
9	Trujillo, Underwood	Development	Oakland, CA	Edit

Figure 39-1. *Running the example application*

The main application database is automatically reseeded when the application starts. There is no seed data for the ASP.NET Core Identity database. Request `http://localhost:5000/users/list` and `http://localhost:5000/roles/list`, and you will see the responses in Figure 39-2, which show the database is empty.

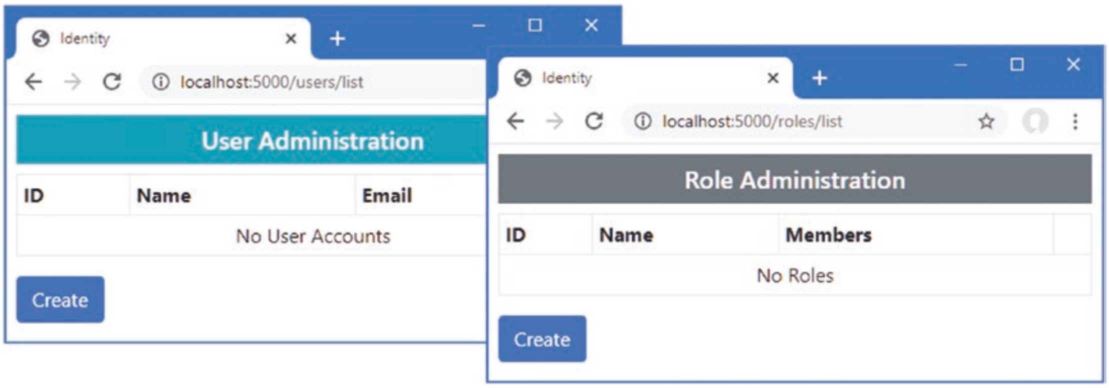


Figure 39-2. *The empty ASP.NET Core Identity database*

Authenticating Users

In the sections that follow, I show you how to add authentication features to the example project so that users can present their credentials and establish their identity to the application.

AUTHENTICATION VS. AUTHORIZATION

It is important to understand the difference between authentication and authorization when working with ASP.NET Core Identity. *Authentication*, often referred to as *AuthN*, is the process of establishing the identity of a user, which the user does by presenting their credentials to the application. In the case of the example application, those credentials are a username and a password. The username is public information, but the password is known only by the user, and when the correct password is presented, the application is able to authenticate the user.

Authorization, often referred to as *AuthZ*, is the process of granting access to application features based on a user's identity. Authorization can be performed only when a user has been authenticated because an application has to know the identity of a user before deciding whether they are entitled to use a specific feature.

Creating the Login Feature

To enforce a security policy, the application must allow users to authenticate themselves, which is done using the ASP.NET Core Identity API. Create the `Pages/Account` folder and add to it a Razor Page named `_Layout.cshtml` with the content shown in Listing 39-3. This layout will provide common content for authentication features.

Listing 39-3. The Contents of the _Layout.cshtml File in the Pages/Account Folder in the Advanced Project

```
<!DOCTYPE html>
<html>
<head>
    <title>Identity</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="m-2">
        @RenderBody()
    </div>
</body>
</html>
```

Add a Razor Page named Login.cshtml to the Pages/Account folder in the Advanced project with the content shown in Listing 39-4.

Listing 39-4. The Contents of the Login.cshtml File in the Pages/Account Folder of the Advanced Project

```
@page
@model LoginModel

<div class="bg-primary text-center text-white p-2"><h4>Log In</h4></div>

<div class="m-1 text-danger" asp-validation-summary="All"></div>

<form method="post">
    <input type="hidden" name="returnUrl" value="@Model.ReturnUrl" />
    <div class="form-group">
        <label>UserName</label>
        <input class="form-control" asp-for="UserName" />
    </div>
    <div class="form-group">
        <label>Password</label>
        <input asp-for="Password" type="password" class="form-control" />
    </div>
    <button class="btn btn-primary" type="submit">Log In</button>
</form>

@functions {

    public class LoginModel : PageModel {
        private SignInManager<IdentityUser> signInManager;

        public LoginModel(SignInManager<IdentityUser> signinMgr) {
            signInManager = signinMgr;
        }

        [BindProperty] [Required]
        public string UserName { get; set; }

        [BindProperty] [Required]
        public string Password { get; set; }

        [BindProperty(SupportsGet = true)]
        public string ReturnUrl { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                Microsoft.AspNetCore.Identity.SignInResult result =
                    await signInManager.PasswordSignInAsync(UserName, Password,
                        false, false);
                if (result.Succeeded) {
                    return Redirect(ReturnUrl ?? "/");
                }
                ModelState.AddModelError("", "Invalid username or password");
            }
            return Page();
        }
    }
}
```

ASP.NET Core Identity provides the `SigninManager<T>` class to manage logins, where the generic type argument `T` is the class that represents users in the application, which is `IdentityUser` for the example application. Table 39-2 describes the `SigninManager<T>` members I use in this chapter.

Table 39-2. *Useful SigninManager<T> Members*

Name	Description
`PasswordSignInAsync(name, password, persist, lockout)`	This method attempts authentication using the specified username and password. The `persist` argument determines whether a successful authentication produces a cookie that persists after the browser is closed. The `lockout` argument determines whether the account should be locked if authentication fails.
`SignOutAsync()`	This method signs out the user.

The Razor Page presents the user with a form that collects a username and a password, which are used to perform authentication with the `PasswordSignInAsync` method, like this:

```
...
Microsoft.AspNetCore.Identity.SignInResult result =
    await signInManager.PasswordSignInAsync(UserName, Password, false, false);
...
```

The result from the `PasswordSignInAsync` methods is a `SignInResult` object, which defines a `Suceeded` property that is `true` if the authentication is successful. (There is also a `SignInResult` class defined in the `Microsoft.AspNetCore.Mvc` namespace, which is why I used a fully qualified class name in the listing.)

Authentication in an ASP.NET Core application is usually triggered when the user tries to access an endpoint that requires authorization, and it is convention to return the user to that endpoint if authentication is successful, which is why the `Login` page defines a `ReturnUrl` property that is used in a redirection if the user has provided valid credentials.

```
...
if (result.Succeeded) {
    return Redirect(ReturnUrl ?? "/");
}
...
```

If the user hasn't provided valid credentials, then a validation message is shown, and the page is redisplayed.

PROTECTING THE AUTHENTICATION COOKIE

The authentication cookie contains the user's identity, and ASP.NET Core trusts that requests containing the cookie originate from the authenticated user. This means you should use HTTPS for production applications that use ASP.NET Core Identity to prevent the cookie from being intercepted by an intermediary. See Part 2 for details of enabling HTTPS in ASP.NET Core.

Inspecting the ASP.NET Core Identity Cookie

When a user is authenticated, a cookie is added to the response so that subsequent requests can be identified as being already authenticated. Add a Razor Page named `Details.cshtml` to the `Pages/Account` folder of the Advanced project with the content shown in Listing 39-5, which displays the cookie when it is present.

Listing 39-5. The Contents of the Details.cshtml File in the Pages/Account Folder of the Advanced Folder

```
@page
@model DetailsModel

<table class="table table-sm table-bordered">
    <tbody>
        @if (Model.Cookie == null) {
            <tr><th class="text-center">No Identity Cookie</th></tr>
        } else {
            <tr>
                <th>Cookie</th>
                <td class="text-break">@Model.Cookie</td>
            </tr>
        }
    </tbody>
</table>

@functions {

    public class DetailsModel : PageModel {

        public string Cookie { get; set; }

        public void OnGet() {
            Cookie = Request.Cookies[".AspNetCore.Identity.Application"];
        }
    }
}
```

The name used for the ASP.NET Core Identity cookie is `.AspNetCore.Identity.Application`, and this page retrieves the cookie from the request and displays its value or a placeholder message if there is no cookie.

Creating a Sign-Out Page

It is important to give users the ability to sign out so they can explicitly delete the cookie, especially if public machines may be used to access the application. Add a Razor Page named `Logout.cshtml` to the `Pages/Account` folder of the `Advanced` folder with the content shown in Listing 39-6.

Listing 39-6. The Contents of the Logout.cshtml File in the Pages/Account Folder in the Advanced Project

```
@page
@model LogoutModel

<div class="bg-primary text-center text-white p-2"><h4>Log Out</h4></div>
<div class="m-2">
    <h6>You are logged out</h6>
    <a asp-page="Login" class="btn btn-secondary">OK</a>
</div>

@functions {

    public class LogoutModel : PageModel {
        private SignInManager<IdentityUser> signInManager;

        public LogoutModel(SignInManager<IdentityUser> signInMgr) {
            signInManager = signInMgr;
        }
        public async Task OnGetAsync() {
            await signInManager.SignOutAsync();
        }
    }
}
```

This page calls the SignOutAsync method described in Table 39-2 to sign the application out of the application. The ASP.NET Core Identity cookie will be deleted so that the browser will not include it in future requests (and invalidated the cookie, so that requests will not be treated as authenticated even if the cookie is used again anyway).

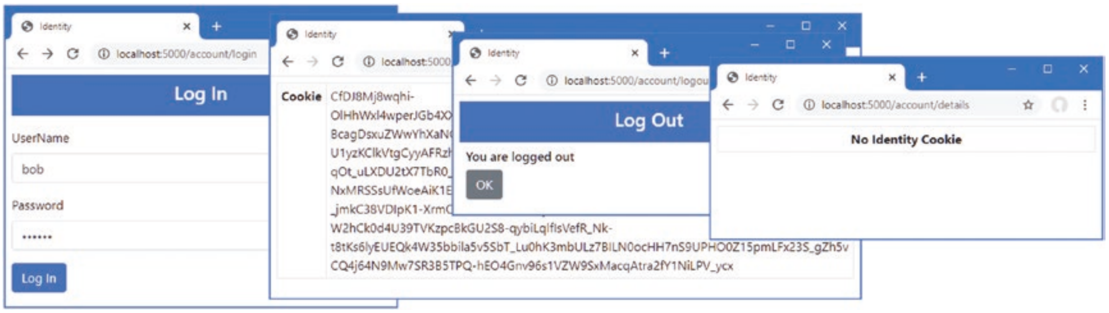
Testing the Authentication Feature

Restart ASP.NET Core and request `http://localhost:5000/users/list`. Click the Create button and fill out the form using the data shown in Table 39-3. Click the Submit button to submit the form and create the user account.

Table 39-3. *The Data Values to Create a User*

Field	Description
`UserName`	bob
`Email`	bob@example.com
`Password`	`secret`

Navigate to `http://localhost:5000/account/login` and authenticate using the username and password from Table 39-3. No return URL has been specified, and you will be redirected to the root URL once you have been authenticated. Request `http://localhost:5000/account/details`, and you will see the ASP.NET Core Identity cookie. Request `http://localhost:5000/account/logout` to log out of the application and return to `http://localhost:5000/account/details` to confirm that the cookie has been deleted, as shown in Figure 39-3.



Cookie	CfDJ8Mj8wqhi- OlHhWxl4wperJGb4X BcagDsxuZWwYhXaN U1yzKClkVtgCyyAFRz qOt_uLXDU2tX7TbR0_ NxMRSSsUfWoeAiK1E _jmkC38VDIpK1-Xrm W2hCk0d4U39TVKzpcBkGU2S8-qybiLqIfIsVefR_Nk- t8tKs6lyEUEQk4W35bbila5v5SbT_Lu0hK3mbULz7BILN0ocHH7nS9UPHO0Z15pmLFx23S_gZh5v CQ4j64N9Mw7SR3B5TPQ-hEO4Gnv96s1VZW9SxMacqAtra2fY1NiLPV_ycx

Figure 39-3. *Authenticating a user*

Enabling the Identity Authentication Middleware

ASP.NET Core Identity provides a middleware component that detects the cookie created by the `SignInManager<T>` class and populates the `HttpContex` object with details of the authenticated user. This provides endpoints with details about the user without needing to be aware of the authentication process or having to deal directly with the cookie created by the authentication process. Listing 39-7 adds the authentication middleware to the example application's request pipeline.

Listing 39-7. Enabling Middleware in the Startup.cs File in the Advanced Folder

```
...
public void Configure(IApplicationBuilder app, DataContext context) {

    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseRouting();

    app.UseAuthentication();

    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute("controllers",
            "controllers/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();

        endpoints.MapFallbackToClientSideBlazor<BlazorWebAssembly.Startup>
            ("/webassembly/{*path:nonfile}", "index.html");

        endpoints.MapFallbackToPage("/_Host");
    });

    app.Map("/webassembly", opts =>
        opts.UseClientSideBlazorFiles<BlazorWebAssembly.Startup>());

    SeedData.SeedDatabase(context);
}
...
```

The middleware sets the value of the `HttpContext.User` property to a `ClaimsPrincipal` object. *Claims* are pieces of information about a user and details of the source of that information, providing a general-purpose approach to describing the information known about a user.

The `ClaimsPrincipal` class is part of .NET Core and isn't directly useful in most ASP.NET Core applications, but there are two nested properties that are useful in most applications, as described in Table 39-4.

Table 39-4. *Useful Nested ClaimsPrincipal Properties*

Name	Description
`ClaimsPrincipal.Identity.Name`	This property returns the username, which will be `null` if there is no user associated with the request.
`ClaimsPrincipal.Identity.IsAuthenticated`	This property returns `true` if the user associated with the request has been authenticated.

The username provided through the `ClaimsPrincipal` object can be used to obtain the ASP.NET Core Identity user object, as shown in Listing 39-8.

Listing 39-8. User Details in the Details.cshtml File in the Pages/Account Folder of the Advanced Project

```
@page
@model DetailsModel

<table class="table table-sm table-bordered">
    <tbody>
        @if (Model.IdentityUser == null) {
            <tr><th class="text-center">No User</th></tr>
        } else {
            <tr><th>Name</th><td>@Model.IdentityUser.UserName</td></tr>
            <tr><th>Email</th><td>@Model.IdentityUser.Email</td></tr>
        }
    </tbody>
</table>

@functions {

    public class DetailsModel : PageModel {
        private UserManager<IdentityUser> userManager;

        public DetailsModel(UserManager<IdentityUser> manager) {
            userManager = manager;
        }

        public IdentityUser IdentityUser { get; set; }

        public async Task OnGetAsync() {
            if (User.Identity.IsAuthenticated) {
                IdentityUser = await userManager.FindByNameAsync(User.Identity.Name);
            }
        }
    }
}
```

The `HttpContext.User` property can be accessed through the `User` convenience property defined by the `PageModel` and `ControllerBase` classes. This Razor Page confirms that there is an authenticated user associated with the request and gets the `IdentityUser` object that describes the user.

Restart ASP.NET Core, request `http://localhost:5000/account/login`, and authenticate using the details in Table 39-3. Request `http://localhost:5000/account/details`, and you will see how the ASP.NET Core Identity middleware enabled in Listing 39-7 has processed the cookie to associate user details with the request, as shown in Figure 39-4.

CONSIDERING TWO-FACTOR AUTHENTICATION

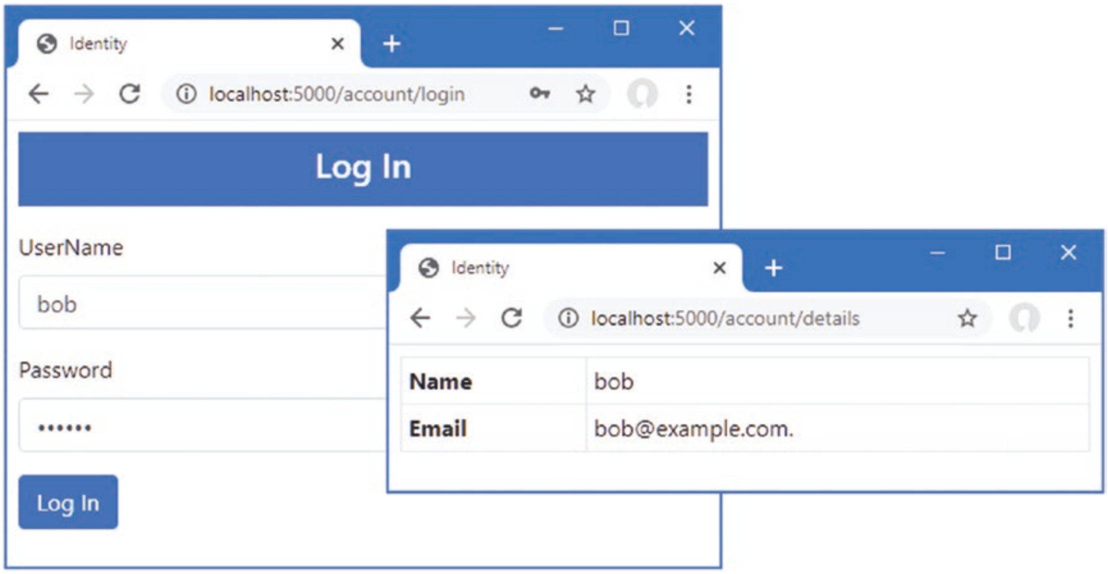
I have performed single-factor authentication in this chapter, which is where the user is able to authenticate using a single piece of information known to them in advance: the password.

ASP.NET Core Identity also supports two-factor authentication, where the user needs something extra, usually something that is given to the user at the moment they want to authenticate. The most common examples are a value from a hardware token or smartphone app or an authentication code that is sent as an e-mail or text message. (Strictly speaking, the two factors can be anything, including fingerprints, iris scans, and voice recognition, although these are options that are rarely required for most web applications.)

Security is increased because an attacker needs to know the user's password *and* have access to whatever provides the second factor, such as an e-mail account or cell phone.

I don't show two-factor authentication in the book for two reasons. The first is that it requires a lot of preparatory work, such as setting up the infrastructure that distributes the second-factor e-mails and texts and implementing the validation logic, all of which is beyond the scope of this book.

The second reason is that two-factor authentication forces the user to remember to jump through an additional hoop to authenticate, such as remembering their phone or keeping a security token nearby, something that isn't always appropriate for web applications. I carried a hardware token of one sort or another for more than a decade in various jobs, and I lost count of the number of times that I couldn't log in to an employer's system because I left the token at home. If you are considering two-factor authentication, then I recommend using one of the many hosted providers that will take care of distributing and managing the second factors for you.



Name	bob
Email	bob@example.com.

Figure 39-4. *Getting details of an authenticated user*

Authorizing Access to Endpoints

Once an application has an authentication feature, user identities can be used to restrict access to endpoints. In the sections that follow, I explain the process for enabling authorization and demonstrate how an authorization policy can be defined.

Applying the Authorization Attribute

The Authorize attribute is used to restrict access to an endpoint and can be applied to individual action or page handler methods or to controller or page model classes, in which case the policy applies to all the methods defined by the class. I want to restrict access to the user and role administration tools created in Chapter 38. When there are multiple Razor Pages or controllers for which the same authorization policy is required, it is a good idea to define a common base class to which the Authorize attribute can be applied because it ensures that you won't accidentally omit the attribute and allow unauthorized access. It is for this reason that I defined the AdminPageModel class and used it as the base for all the administration tool page models in Chapter 38. Listing 39-9 applies the Authorize attribute to the AdminPageModel class to create the authorization policy.

Listing 39-9. Applying an Attribute in the AdminPageModel.cs File in the Pages Folder in the Advanced Project

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authorization;

namespace Advanced.Pages {

    [Authorize(Roles="Admins")]
    public class AdminPageModel : PageModel {

    }
}
```

The Authorize attribute can be applied without arguments, which restricts access to any authenticated user. The Roles argument is used to further restrict access to users who are members of specific roles, which are expressed as a comma-separated list. The attribute in this listing restricts access to users assigned to the Admins role. The authorization restrictions are inherited, which means that applying the attribute to the base class restricts access to all the Razor Pages created to manage users and roles in Chapter 38.

■ **Note** If you want to restrict access to most, but not all, of the action methods in a controller, then you can apply the `Authorize` attribute to the controller class and the `AllowAnonymous` attribute to just the action methods for which authenticated access is required.

Enabling the Authorization Middleware

The authorization policy is enforced by a middleware component, which must be added to the application's request pipeline, as shown in Listing 39-10.

Listing 39-10. Adding Middleware in the Startup.cs File in the Advanced Project

```
...
public void Configure(IApplicationBuilder app, DataContext context) {

    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute("controllers",
            "controllers/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();

        endpoints.MapFallbackToClientSideBlazor<BlazorWebAssembly.Startup>
            ("/webassembly/{*path:nonfile}", "index.html");

        endpoints.MapFallbackToPage("/_Host");
    });

    app.Map("/webassembly", opts =>
        opts.UseClientSideBlazorFiles<BlazorWebAssembly.Startup>());

    SeedData.SeedDatabase(context);
}
...
```

The `UseAuthorization` method must be called between the `UseRouting` and `UseEndpoints` methods and after the `UseAuthentication` method has been called. This ensures that the authorization component can access the user data and inspect the authorization policy after the endpoint has been selected but before the request is handled.

Creating the Access Denied Endpoint

The application must deal with two different types of authorization failure. If no user has been authenticated when a restricted endpoint is requested, then the authorization middleware will return a challenge response, which will trigger a redirection to the login page so the user can present their credentials and prove they should be able to access the endpoint.

But if an authenticated user requests a restricted endpoint and doesn't pass the authorization checks, then an access denied response is generated so the application can display a suitable warning to the user. Add a Razor Page named `AccessDenied.cshtml` to the `Pages/Account` folder of the `Advanced` folder with the content shown in Listing 39-11.

Listing 39-11. The AccessDenied.cshtml File in the Pages/Account Folder of the Advanced Project

```
@page

<h4 class="bg-danger text-white text-center p-2">Access Denied</h4>

<div class="m-2">
    <h6>You are not authorized for this URL</h6>
    <a class="btn btn-outline-danger" href="/">OK</a>
    <a class="btn btn-outline-secondary" asp-page="Logout">Logout</a>
</div>
```

This page displays a warning message to the user, with a button that navigates to the root URL. There is typically little the user can do to resolve authorization failures without administrative intervention, and my preference is to keep the access denied response as simple as possible.

Creating the Seed Data

In Listing 39-9, I restricted access to the user and role administration tools, so they can be accessed only by users in the Admin role. There is no such role in the database, which creates a problem: I am locked out of the administration tools because there is no authorized account that will let me create the role.

I could have created an administration user and role before applying the Authorize attribute, but that complicates deploying the application, when making code changes should be avoided. Instead, I am going to create seed data for ASP.NET Core Identity to ensure there will always be at least one account that can be used to access the user and role management tools. Add a class file named IdentitySeedData.cs to the Models folder in the Advanced project and use it to define the class shown in Listing 39-12.

Listing 39-12. The Contents of the IdentitySeedData.cs File in the Models Folder of the Advanced Project

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace Advanced.Models {
    public class IdentitySeedData {

        public static void CreateAdminAccount(IServiceProvider serviceProvider,
                IConfiguration configuration) {
            CreateAdminAccountAsync(serviceProvider, configuration).Wait();
        }

        public static async Task CreateAdminAccountAsync(IServiceProvider
                serviceProvider, IConfiguration configuration) {

            serviceProvider = serviceProvider.CreateScope().ServiceProvider;

            UserManager<IdentityUser> userManager =
                serviceProvider.GetRequiredService<UserManager<IdentityUser>>();
            RoleManager<IdentityRole> roleManager =
                serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();

            string username = configuration["Data:AdminUser:Name"] ?? "admin";
            string email
                = configuration["Data:AdminUser:Email"] ?? "admin@example.com";
            string password = configuration["Data:AdminUser:Password"] ?? "secret";
            string role = configuration["Data:AdminUser:Role"] ?? "Admins";
            if (await userManager.FindByNameAsync(username) == null) {
                if (await roleManager.FindByNameAsync(role) == null) {
                    await roleManager.CreateAsync(new IdentityRole(role));
                }

                IdentityUser user = new IdentityUser {
                    UserName = username,
                    Email = email
                };

                IdentityResult result = await userManager
                    .CreateAsync(user, password);
                if (result.Succeeded) {
                    await userManager.AddToRoleAsync(user, role);
                }
            }
        }
    }
}
```

The `UserManager<T>` and `RoleManager<T>` services are scoped, which means I need to create a new scope before requesting the services since the seeding will be done when the application starts. The seeding code creates a user account that is assigned to a role. The values for the seed data are read from the application's configuration with fallback values, making it easy to configure the seeded account without needing a code change. Listing 39-13 adds a statement to the `Startup` class so that the database is seeded when the application starts.

■ **Caution** Putting passwords in code files or plain-text configuration files means you must make it part of your deployment process to change the default account's password when you deploy the application and initialize a new database for the first time. You can also use the user secrets feature to keep sensitive data outside of the project.

Listing 39-13. Seeding Identity in the Startup.cs File in the Advanced Project

```
...
public void Configure(IApplicationBuilder app, DataContext context) {

    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute("controllers",
            "controllers/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();

        endpoints.MapFallbackToClientSideBlazor<BlazorWebAssembly.Startup>
            ("/webassembly/{*path:nonfile}", "index.html");

        endpoints.MapFallbackToPage("/_Host");
    });
    app.Map("/webassembly", opts =>
        opts.UseClientSideBlazorFiles<BlazorWebAssembly.Startup>());

    SeedData.SeedDatabase(context);
    IdentitySeedData.CreateAdminAccount(app.ApplicationServices, Configuration);
}
...
```

Testing the Authentication Sequence

Restart ASP.NET Core and request `http://localhost:5000/account/logout` to ensure that no user is logged in to the application. Without logging in, request `http://localhost:5000/users/list`. The endpoint that will be selected to handle the request requires authentication, and the login prompt will be shown since there is no authenticated user associated with the request. Authenticate with the username `bob` and the password `secret`. This user doesn't have access to the restricted endpoint, and the access denied response will be shown, as illustrated by Figure 39-5.

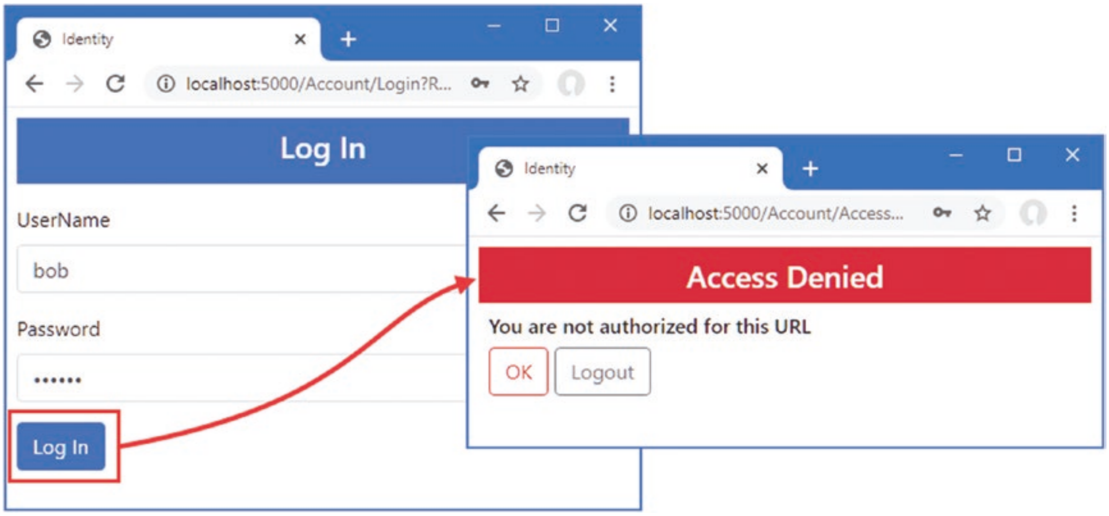
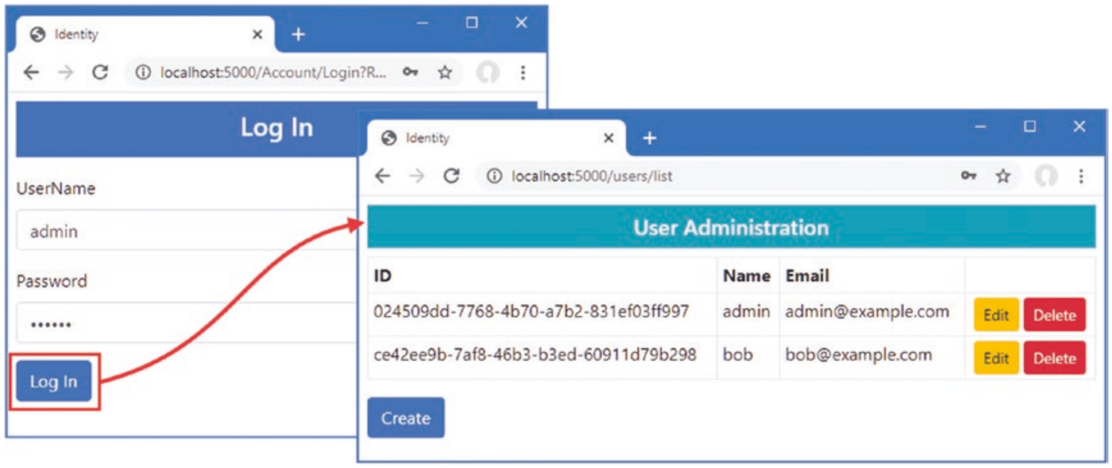

Identity
localhost:5000/Account/Login?R...
Log In
UserName
bob
Password
••••••
Log In
Identity
localhost:5000/Account/Access...
Access Denied
You are not authorized for this URL
OK
Logout

Figure 39-5. *A user without authorization*

Click the Logout button and request `http://localhost:5000/users/list` again, which will lead to the login prompt being displayed. Authenticate with the username `admin` and the password `secret`. This is the user account created by the seed data and that is a member of the role specified by the `Authorize` attribute. The user passes the authorization check, and the requested Razor Page is displayed, as shown in Figure 39-6.



ID	Name	Email	
024509dd-7768-4b70-a7b2-831ef03ff997	admin	admin@example.com	Edit Delete
ce42ee9b-7af8-46b3-b3ed-60911d79b298	bob	bob@example.com	Edit Delete

***Figure 39-6.** A user with authorization*

CHANGING THE AUTHORIZATION URLS

The `/Account/Login` and `/Account/AccessDenied` URLs are the defaults used by ASP.NET Core authorization files. These can be changed in the `Startup` class using the options pattern, like this:

```
...
services.Configure<CookieAuthenticationOptions>(
       IdentityConstants.ApplicationScheme,
    opts => {
        opts.LoginPath = "/Authenticate";
        opts.AccessDeniedPath = "/NotAllowed";
    });
...
```

Configuration is performed using the `CookieAuthenticationOptions` class, defined in the `Microsoft.AspNetCore.Authentication.Cookies` namespace. The `LoginPath` property is used to specify the path to which browsers will be redirected when an unauthenticated user attempts to access a restricted endpoint. The `AccessDeniedPath` property is used to specify the path when an authenticated user attempts to access a restricted endpoint and does not have authorization.

Authorizing Access to Blazor Applications

The simplest way to protect Blazor applications is to restrict access to the action method or Razor Page that acts as the entry point. In Listing 39-14, I added the `Authorize` attribute to the page model class for the `_Host` page, which is the entry point for the Blazor application in the example project.

UNDERSTANDING OAUTH AND IDENTITYSERVER

If you read the Microsoft documentation, you will be left with the impression that you need to use a third-party server called `IdentityServer` (`http://identityserver.io`) to authenticate web services.

`IdentityServer` is a high-quality open source package that provides authentication and authorization services, with paid-for options for add-ons and support. `IdentityServer` provides support for `OAuth`, which is a standard for managing authentication and authorization and provides packages for a range of client-side frameworks.

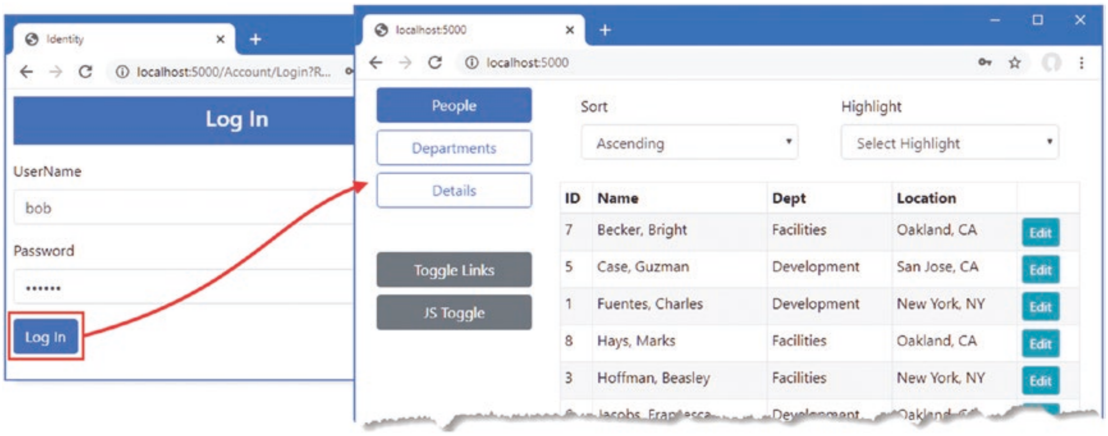
What the Microsoft documentation is saying—albeit awkwardly—is that Microsoft has used `IdentityServer` in the project templates that include authentication for web services. If you create an Angular or React project using an ASP.NET Core template provided by Microsoft, you will find that the authentication has been implemented using `IdentityServer`.

Authentication is complex, and `IdentityServer` can be difficult to set up correctly. I like `IdentityServer`, but it is not essential and is not required by most projects. `IdentityServer` may be useful if your project needs to support complex authentication scenarios, but my advice is not to rush into using third-party authentication servers until they are essential.

Listing 39-14. Applying an Attribute in the _Host.cshtml File in the Pages Folder of the Advanced Project

```
@page "/"
@{ Layout = null; }
@model HostModel
@using Microsoft.AspNetCore.Authorization

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <base href="~/" />
</head>
<body>
    <div class="m-2">
        <component type="typeof(Advanced.Blazor.Routed)" render-mode="Server" />
    </div>
    <script src="_framework/blazor.server.js"></script>
    <script src="~/interop.js"></script>
</body>
</html>

@functions {

    [Authorize]
    public class HostModel : PageModel {}
}
```

This has the effect of preventing unauthenticated users from accessing the Blazor application. Request `http://localhost:5000/account/logout` to ensure the browser doesn't have an authentication cookie and then request `http://localhost:5000`. This request will be handled by the `_Host` page, but the authorization middleware will trigger the redirection to the login prompt. Authenticate with the username bob and the password `secret`, and you will be granted access to the Blazor application, as shown in Figure 39-7.



UserName
bob

Password
••••••

Sort: Ascending

Highlight: Select Highlight

ID	Name	Dept	Location
7	Becker, Bright	Facilities	Oakland, CA
5	Case, Guzman	Development	San Jose, CA
1	Fuentes, Charles	Development	New York, NY
8	Hays, Marks	Facilities	Oakland, CA
3	Hoffman, Beasley	Facilities	New York, NY

Figure 39-7. *Restricting access to the Blazor endpoint*

Performing Authorization in Blazor Components

Restricting access to the endpoint is an effective technique, but it applies the same level of authorization to all the Blazor functionality. For applications that require more granular restrictions, Blazor provides the `AuthorizeRouteView` component, which allows different content to be displayed for authorized and unauthorized when components are managed using URL routing. Listing 39-15 adds the `AuthorizeRouteView` to the routing component in the example application.

Listing 39-15. Adding a Component in the Routed.razor File in the Blazor Folder of the Advanced Project

```
@using Microsoft.AspNetCore.Components.Authorization

<Router AppAssembly="typeof(Startup).Assembly">
    <Found>
        <AuthorizeRouteView RouteData="@context" DefaultLayout="typeof(NavLayout)">
           <NotAuthorized Context="authContext">
                <h4 class="bg-danger text-white text-center p-2">Not Authorized </h4>
                <div class="text-center">
                    You may need to log in as a different user
                </div>
            </NotAuthorized>
        </AuthorizeRouteView>
    </Found>
    <NotFound>
        <h4 class="bg-danger text-white text-center p-2">
            Not Matching Route Found
        </h4>
    </NotFound>
</Router>
```

The `NotAuthorized` section is used to define the content that will be presented to users when they attempt to access a restricted resource. To demonstrate this feature, I am going to restrict access to the `DepartmentList` component to users assigned to the `Admins` role, as shown in Listing 39-16.

Listing 39-16. Restricting Access in the DepartmentList.cshtml File in the Blazor Folder in the Advanced Project

```
@page "/departments"
@page "/depts"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize(Roles = "Admins")]
```

```
<CascadingValue Name="BgTheme" Value="Theme" IsFixed="false" >
    <TableTemplate RowType="Department" RowData="Departments"
        Highlight="@(d => d.Name)"
        SortDirection="@(d => d.Name)">
        <Header>
            <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
        </Header>
        <RowTemplate Context="d">
            <td>@d.Departmentid</td>
            <td>@d.Name</td>
            <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
            <td>
                @(String.Join(", ",
                    d.People.Select(p => p.Location.City).Distinct()))
            </td>
        </RowTemplate>
    </TableTemplate>
</CascadingValue>

<SelectFilter Title="@("Theme")" Values="Themes" @bind-SelectedValue="Theme" />

<button class="btn btn-primary" @onclick="HandleClick">People</button>

@code {

    [Inject]
    public DataContext Context { get; set; }

    public IEnumerable<Department> Departments => Context.Departments
            .Include(d => d.People).ThenInclude(p => p.Location);

    public string Theme { get; set; } = "info";
    public string[] Themes = new string[] { "primary", "info", "success" };

    [Inject]
    public NavigationManager NavManager { get; set; }

    public void HandleClick() => NavManager.NavigateTo("/people");
}
```

I have used the `@attribute` directive to apply the `Authorize` attribute to the component. Restart ASP.NET Core and request `http://localhost:5000/account/logout` to remove the authentication cookie and then request `http://localhost:5000`. When prompted, authenticate with the username `bob` and the `password` secret. You will see the Blazor application, but when you click the Departments button, you will see the authorization content defined in Listing 39-15, as shown in Figure 39-8. Log out again and log in as `admin` with the password `secret`, and you will be able to use the restricted component.

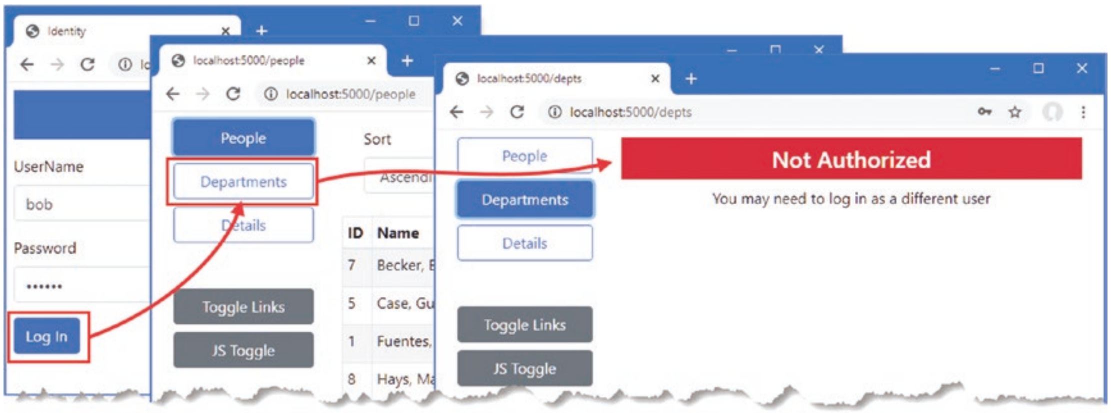


Figure 39-8. *Using authorization in a Blazor application*

Displaying Content to Authorized Users

The AuthorizeView component is used to restrict access to sections of content rendered by a component. In Listing 39-17, I have changed the authorization for the DepartmentList component so that any authenticated user can access the page and use the AuthorizeView component so that the contents of the Locations column in the table is shown only to users assigned to the Admins group.

Listing 39-17. Selective Content in the DepartmentList.razor File in the Blazor Folder in the Advanced Project

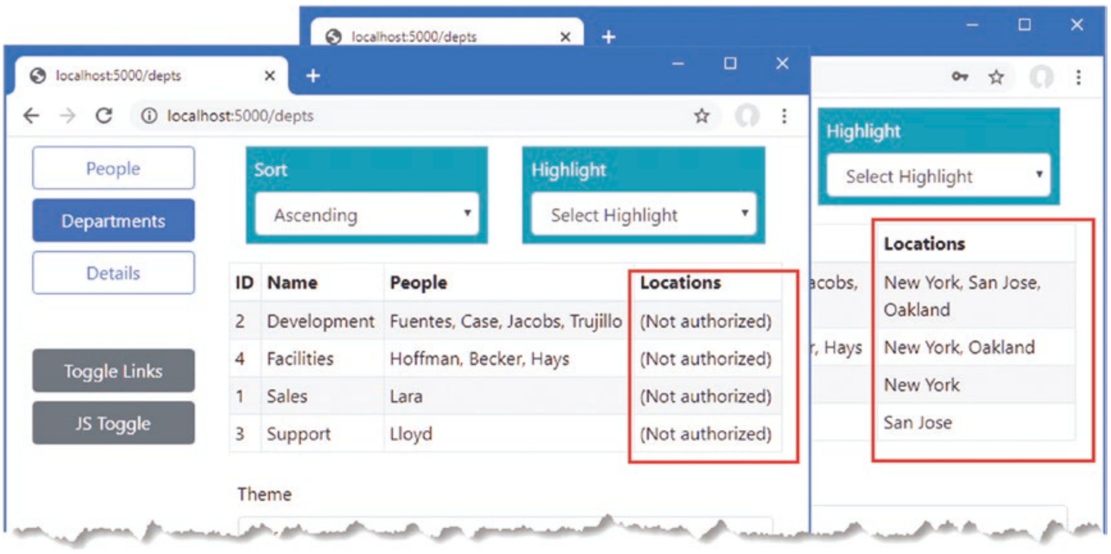
```
@page "/departments"
@page "/depts"
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@attribute [Authorize]

<CascadingValue Name="BgTheme" Value="Theme" IsFixed="false" >
    <TableTemplate RowType="Department" RowData="Departments"
        Highlight="@(d => d.Name)"
        SortDirection="@(d => d.Name)">
        <Header>
            <tr><th>ID</th><th>Name</th><th>People</th><th>Locations</th></tr>
        </Header>
        <RowTemplate Context="d">
            <td>@d.Departmentid</td>
            <td>@d.Name</td>
            <td>@(String.Join(", ", d.People.Select(p => p.Surname)))</td>
            <td>
                <AuthorizeView Roles="Admins">
                    <Authorized>
                        @(String.Join(", ",
                            d.People.Select(p => p.Location.City).Distinct()))
                    </Authorized>
                    <NotAuthorized>
                        (Not authorized)
                    </NotAuthorized>
                </AuthorizeView>
            </td>
        </RowTemplate>
    </TableTemplate>
</CascadingValue>
```

```
<SelectFilter Title="@("Theme")" Values="Themes" @bind-SelectedValue="Theme" />

<button class="btn btn-primary" @onclick="HandleClick">People</button>

@code {

    // ...statements omitted for brevity...
}
```

The `AuthorizeView` component is configured with the `Roles` property, which accepts a comma-separated list of authorized roles. The `Authorized` section contains the content that will be shown to authorized users. The `NotAuthorized` section contains the content that will be shown to unauthorized users.

■ **Tip** You can omit the `NotAuthorized` section if you don't need to show content to unauthorized users.

Restart ASP.NET Core and authenticate as `bob`, with password `secret`, before requesting `http://localhost:5000/depts`. This user is not authorized to see the contents of the `Locations` column, as shown in Figure 39-9. Authenticate as `admin`, with password `secret`, and request `http://localhost:5000/depts` again. This time the user is a member of the `Admins` role and passes the authorization checks, also shown in Figure 39-9.



ID	Name	People	Locations
2	Development	Fuentes, Case, Jacobs, Trujillo	(Not authorized)
4	Facilities	Hoffman, Becker, Hays	(Not authorized)
1	Sales	Lara	(Not authorized)
3	Support	Lloyd	(Not authorized)

Locations
New York, San Jose, Oakland
New York, Oakland
New York
San Jose

Figure 39-9. Selectively displaying content based on authorization

Authenticating and Authorizing Web Services

The authorization process in the previous section relies on being able to redirect the client to a URL that allows the user to enter their credentials. A different approach is required when adding authentication and authorization to a web service because there is no option to present the user with an HTML form to collect their credentials. The first step in adding support for web services authentication is to disable the redirections so that the client will receive HTTP error responses when attempting to request an endpoint that requires authentication. Add a class file named `CookieAuthenticationExtensions.cs` to the `Advanced` folder and use it to define the extension method shown in Listing 39-18.

Listing 39-18. The Contents of the CookieAuthenticationExtensions.cs File in the Advanced Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Authentication.Cookies {
    public static class CookieAuthenticationExtensions {

        public static void DisableRedirectForPath(
            this CookieAuthenticationEvents events,
            Expression<Func<CookieAuthenticationEvents,
                Func<RedirectContext<CookieAuthenticationOptions>, Task>>> expr,
            string path, int statuscode) {

            string propertyName = ((MemberExpression)expr.Body).Member.Name;
            var oldHandler = expr.Compile().Invoke(events);

            Func<RedirectContext<CookieAuthenticationOptions>, Task> newHandler
                    = context => {
                if (context.Request.Path.StartsWithSegments(path)) {
                    context.Response.StatusCode = statuscode;
                } else {
                    oldHandler(context);
                }
                return Task.CompletedTask;
            };

            typeof(CookieAuthenticationEvents).GetProperty(propertyName)
                .SetValue(events, newHandler);
        }
    }
}
```

This code is hard to follow. ASP.NET Core provides the `CookieAuthenticationOptions` class, which is used to configure cookie-based authentication. The `CookieAuthenticationOptions.Events` property returns a `CookieAuthenticationEvents` object, which is used to set the handlers for the events triggered by the authentication system, including the redirections that occur when the user requests unauthorized content. The extension methods in Listing 39-18 replaces the default handler for an event with one that performs redirection only if the request doesn't start with a specified path string. Listing 39-19 uses the extension method to replace the `OnRedirectToLogin` and `OnRedirectToAccessDenied` handlers so that redirections are not performed when the request path starts with `/api`.

Listing 39-19. Preventing Redirection in the Startup.cs File in the Advanced Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Advanced.Models;
using Microsoft.AspNetCore.ResponseCompression;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Authentication.Cookies;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddServerSideBlazor();
            services.AddSingleton<Services.ToggleService>();

            services.AddResponseCompression(opts => {
                opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
                    new[] { "application/octet-stream" });
            });

            services.AddDbContext<IdentityContext>(opts =>
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:IdentityConnection"]));
            services.AddIdentity<IdentityUser, IdentityRole>()
                .AddEntityFrameworkStores<IdentityContext>();

            services.Configure<IdentityOptions>(opts => {
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric = false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
                opts.User.RequireUniqueEmail = true;
                opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
            });

            services.AddAuthentication(opts => {
                opts.DefaultScheme =
                    CookieAuthenticationDefaults.AuthenticationScheme;
                opts.DefaultChallengeScheme =
                     CookieAuthenticationDefaults.AuthenticationScheme;
            }).AddCookie(opts => {
                opts.Events.DisableRedirectForPath(e => e.OnRedirectToLogin,
                     "/api", StatusCodes.Status401Unauthorized);
                opts.Events.DisableRedirectForPath(e => e.OnRedirectToAccessDenied,
                    "/api", StatusCodes.Status403Forbidden);
            });
        }
        public void Configure(IApplicationBuilder app, DataContext context) {

            // ...statements omitted for brevity...
        }
    }
}
```

The AddAuthentication method is used to select cookie-based authentication and is chained with the AddCookie method to replace the event handlers that would otherwise trigger redirections.

Building a Simple JavaScript Client

To demonstrate how to perform authentication with web services, I am going to create a simple JavaScript client that will consume data from the Data controller in the example project.

■ **Tip** You don't have to be familiar with JavaScript to follow the examples in this part of the chapter. It is the server-side code that is important and the way it supports authentication by the client so that it can access the web service.

Add an HTML Page called webclient.html to the wwwroot folder of the Advanced project with the elements shown in Listing 39-20.

Listing 39-20. The Contents of the webclient.html File in the wwwroot Folder of the Advanced Project

```
<!DOCTYPE html>
<html>
<head>
    <title>Web Service Authentication</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <script type="text/javascript" src="webclient.js"></script>
</head>
<body>
    <div id="controls" class="m-2"></div>
    <div id="data" class="m-2 p-2">
        No data
    </div>
</body>
</html>
```

Add a JavaScript file named webclient.js to the wwwroot of the Advanced project with the content shown in Listing 39-21.

Listing 39-21. The Contents of the webclient.js File in the wwwroot Folder of the Advanced Project

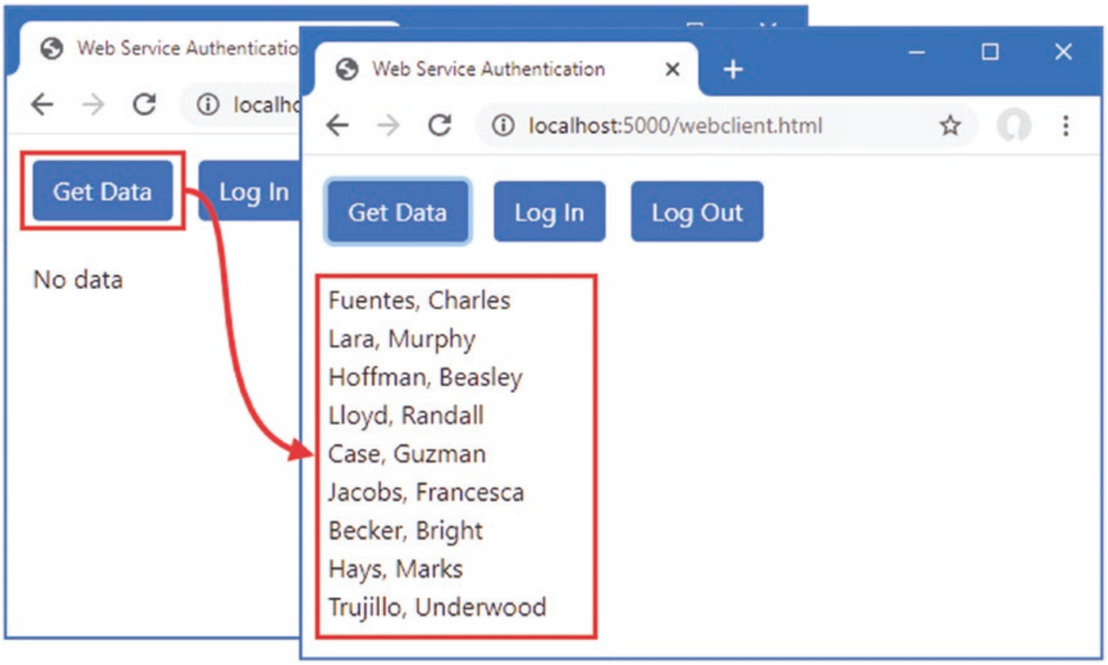
```
const username = "bob";
const password = "secret";

window.addEventListener("DOMContentLoaded", () => {
    const controlDiv = document.getElementById("controls");
    createButton(controlDiv, "Get Data", getData);
    createButton(controlDiv, "Log In", login);
    createButton(controlDiv, "Log Out", logout);
});

function login() {
    // do nothing
}
```

```
function logout() {
    // do nothing
}

async function getData() {
    let response = await fetch("/api/people");
    if (response.ok) {
        let jsonData = await response.json();
        displayData(...jsonData.map(item => `${item.surname}, ${item.firstname}`));
    } else {
        displayData(`Error: ${response.status}: ${response.statusText}`);
    }
}

function displayData(...items) {
    const dataDiv = document.getElementById("data");
    dataDiv.innerHTML = "";
    items.forEach(item => {
        const itemDiv = document.createElement("div");
        itemDiv.innerText = item;
        itemDiv.style.wordWrap = "break-word";
        dataDiv.appendChild(itemDiv);
    })
}

function createButton(parent, label, handler) {
    const button = document.createElement("button");
    button.classList.add("btn", "btn-primary",  "m-2");
    button.innerText = label;
    button.onclick = handler;
    parent.appendChild(button);
}
```

This code presents the user with Get Data and Log In and Log Out buttons. Clicking the Get Data button sends an HTTP request using the Fetch API, processes the JSON result, and displays a list of names. The other buttons do nothing, but I'll use them in later examples to authenticate with the ASP.NET Core application using the hardwired credentials in the JavaScript code.

■ **Caution** This is just a simple client to demonstrate server-side authentication features. If you need to write a JavaScript client, then consider a framework such as Angular or React. Regardless of how you build your clients, do not include hardwired credentials in the JavaScript files.

Request `http://localhost:5000/webclient.html`, and click the Get Data button. The JavaScript client will send an HTTP request to the `Data` controller and display the results, as shown in Figure 39-10.


Web Service Authenticatio
localh
Get Data
Log In
No data
Web Service Authentication
localhost:5000/webclient.html
Get Data
Log In
Log Out

Fuentes, Charles
Lara, Murphy
Hoffman, Beasley
Lloyd, Randall
Case, Guzman
Jacobs, Francesca
Becker, Bright
Hays, Marks
Trujillo, Underwood

Figure 39-10. *A simple web client*

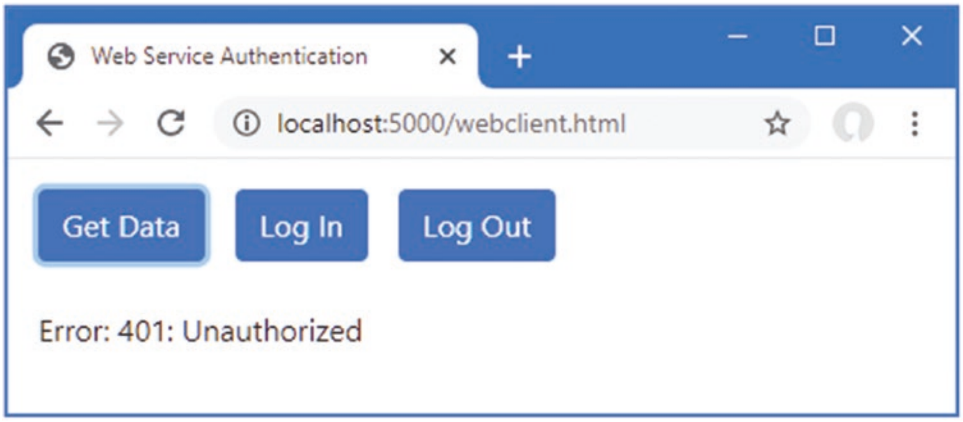
Restricting Access to the Web Service

The standard authorization features are used to restrict access to web service endpoints, and in Listing 39-22, I have applied the `Authorize` attribute to the `DataController` class.

Listing 39-22. Applying an Attribute in the DataController.cs File in the Controllers Folder of the Advanced Project

```
using Advanced.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Advanced.Controllers {

    [ApiController]
    [Route("/api/people")]
    [Authorize]
    public class DataController : ControllerBase {
        private DataContext context;

        // ...methods omitted for brevity...
    }
}
```

Restart ASP.NET Core and request `http://localhost:5000/account/logout` to ensure that the JavaScript client doesn't use an authentication cookie from a previous example. Request `http://localhost:5000/webclient.html` to load the JavaScript client and click the Get Data button to send the HTTP request. The server will respond with a `401 Unauthorized` response, as shown in Figure 39-11.



Error: 401: Unauthorized

Figure 39-11. *An unauthorized request*

Using Cookie Authentication

The simplest way to implement authentication is to rely on the standard ASP.NET Core cookies demonstrated in previous sections. Add a class file named `ApiAccountController.cs` to the `Controllers` folder of the `Advanced` project and use it to define the controller shown in Listing 39-23.

Listing 39-23. The Contents of the ApiAccountController.cs File in the Controllers Folder of the Advanced Project

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;

namespace Advanced.Controllers {

    [ApiController]
    [Route("/api/account")]
    public class ApiAccountController : ControllerBase {
        private SignInManager<IdentityUser> signinManager;

        public ApiAccountController(SignInManager<IdentityUser> mgr) {
            signinManager = mgr;
        }

        [HttpPost("login")]
        public async Task<IActionResult> Login([FromBody]Credentials creds) {
            Microsoft.AspNetCore.Identity.SignInResult result
                = await signinManager.PasswordSignInAsync(creds.Username,
                     creds.Password, false, false);
            if (result.Succeeded) {
                return Ok();
            }
            return Unauthorized();
        }

        [HttpPost("logout")]
        public async Task<IActionResult> Logout() {
            await signinManager.SignOutAsync();
            return Ok();
        }
        public class Credentials {
            [Required]
            public string Username { get; set; }
            [Required]
            public string Password { get; set; }
        }
    }
}
```

This web service controller defines actions that allow clients to log in and log out. The response for a successful authentication request will contain a cookie that the browser will automatically include in requests made by the JavaScript client.

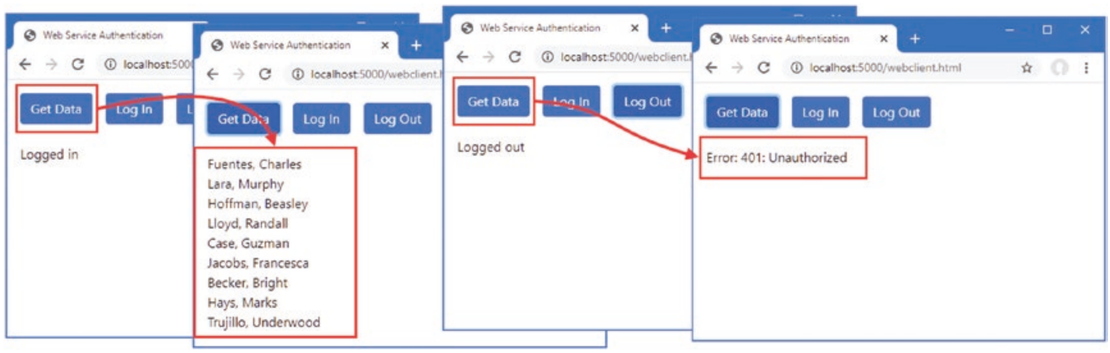
Listing 39-24 adds support to the simple JavaScript client for authenticating using the action methods defined in Listing 39-23.

Listing 39-24. Adding Authentication in the webclient.js File in the wwwroot Folder of the Advanced Project

```
const username = "bob";
const password = "secret";

window.addEventListener("DOMContentLoaded", () => {
    const controlDiv = document.getElementById("controls");
    createButton(controlDiv, "Get Data", getData);
    createButton(controlDiv, "Log In", login);
    createButton(controlDiv, "Log Out", logout);
});

async function login() {
    let response = await fetch("/api/account/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ username: username, password: password })
    });
    if (response.ok) {
        displayData("Logged in");
    } else {
        displayData(`Error: ${response.status}: ${response.statusText}`);
    }
}

async function logout() {
    let response = await fetch("/api/account/logout", {
        method: "POST"
    });
    if (response.ok) {
        displayData("Logged out");
    } else {
        displayData(`Error: ${response.status}: ${response.statusText}`);
    }
}

async function getData() {
    let response = await fetch("/api/people");
    if (response.ok) {
        let jsonData = await response.json();
        displayData(...jsonData.map(item => `${item.surname}, ${item.firstname}`));
    } else {
        displayData(`Error: ${response.status}: ${response.statusText}`);
    }
}
```

```
function displayData(...items) {
    const dataDiv = document.getElementById("data");
    dataDiv.innerHTML = "";
    items.forEach(item => {
        const itemDiv = document.createElement("div");
        itemDiv.innerText = item;
        itemDiv.style.wordWrap = "break-word";
        dataDiv.appendChild(itemDiv);
    })
}

function createButton(parent, label, handler) {
    const button = document.createElement("button");
    button.classList.add("btn", "btn-primary",  "m-2");
    button.innerText = label;
    button.onclick = handler;
    parent.appendChild(button);
}
```

Restart ASP.NET Core, request `http://localhost:5000/webclient.html`, and click the Login In button. Wait for the message confirming authentication and then click the Get Data button. The browser includes the authentication cookie, and the request passes the authorization checks. Click the Log Out button and then click Get Data again. No cookie is used, and the request fails. Figure 39-12 shows both requests.



Logged in

Fuentes, Charles
Lara, Murphy
Hoffman, Beasley
Lloyd, Randall
Case, Guzman
Jacobs, Francesca
Becker, Bright
Hays, Marks
Trujillo, Underwood

Logged out

Error: 401: Unauthorized

Figure 39-12. *Using cookie authentication*

Using Bearer Token Authentication

Not all web services will be able to rely on cookies because not all clients can use then. An alternative is to use a bearer token, which is a string that clients are given and is included in the requests they send to the web service. Clients don't understand the meaning of the token—which is said to be *opaque*—and just use whatever token the server provides.

I am going to demonstrate authentication using a JSON Web Token (JWT), which provides the client with an encrypted token that contains the authenticated username. The client is unable to decrypt or modify the token, but when it is included in a request, the ASP.NET Core server decrypts the token and uses the name it contains as the identity of the user. The JWT format is described in detail at `https://tools.ietf.org/html/rfc7519`.

■ **Caution** ASP.NET Core will trust that any request that includes the token originates from the authenticated user. Just as when using cookies, production applications should use HTTPS to prevent tokens from being intercepted and reused.

Preparing the Application

Open a new PowerShell command prompt, navigate to the Advanced project folder, and run the commands shown in Listing 39-25 to add the packages for JWT to the project.

Listing 39-25. Installing the NuGet Package

```
dotnet add package System.IdentityModel.Tokens.Jwt --version 5.6.0
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer --version 3.1.1
```

JWT requires a key that is used to encrypt and decrypt tokens. Add the configuration setting shown in Listing 39-26 to the appsettings.json file. If you use JWT in a real application, ensure you change the key.

Listing 39-26. Adding a Setting in the appsettings.json File in the Advanced Project

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore": "Information",
      "Microsoft.AspNetCore.Authentication": "Debug"

    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "PeopleConnection": "Server=(localdb)\\MSSQLLocalDB;Database=People;MultipleActiveResultSets=True",
    "IdentityConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;MultipleActiveResultSets=True"
  },
  "jwtSecret": "apress_jwt_secret"
}
```

Creating Tokens

The client will send an HTTP request that contains user credentials and will receive a JWT in response. Listing 39-27 adds an action method to the ApiAccount controller that receives the credentials, validates them, and generates tokens.

Listing 39-27. Generating Tokens in the ApiAccountController.cs File in the Controllers Folder of the Advanced Project

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Text;
using System.Security.Claims;
using System;
using Microsoft.Extensions.Configuration;
```

```
namespace Advanced.Controllers {

    [ApiController]
    [Route("/api/account")]
    public class ApiAccountController : ControllerBase {
        private SignInManager<IdentityUser> signinManager;
        private UserManager<IdentityUser> userManager;
        private IConfiguration configuration;

        public ApiAccountController(SignInManager<IdentityUser> mgr,
                 UserManager<IdentityUser> usermgr, IConfiguration config) {
            signinManager = mgr;
            userManager = usermgr;
            configuration = config;
        }

        [HttpPost("login")]
        public async Task<IActionResult> Login([FromBody]Credentials creds) {
            Microsoft.AspNetCore.Identity.SignInResult result
                = await signinManager.PasswordSignInAsync(creds.Username,
                    creds.Password, false, false);
            if (result.Succeeded) {
                return Ok();
            }
            return Unauthorized();
        }

        [HttpPost("logout")]
        public async Task<IActionResult> Logout() {
            await signinManager.SignOutAsync();
            return Ok();
        }

        [HttpPost("token")]
        public async Task<IActionResult> Token([FromBody]Credentials creds) {
            if (await CheckPassword(creds)) {
                JwtSecurityTokenHandler handler = new JwtSecurityTokenHandler();
                byte[] secret = Encoding.ASCII.GetBytes(configuration["jwtSecret"]);
                SecurityTokenDescriptor descriptor = new SecurityTokenDescriptor {
                    Subject = new ClaimsIdentity(new Claim[] {
                        new Claim(ClaimTypes.Name, creds.Username)
                    }),
                    Expires = DateTime.UtcNow.AddHours(24),
                    SigningCredentials = new SigningCredentials(
                        new SymmetricSecurityKey(secret),
                            SecurityAlgorithms.HmacSha256Signature)
                };
                SecurityToken token = handler.CreateToken(descriptor);
                return Ok(new {
                    success = true,
                    token = handler.WriteToken(token)
                });
            }
            return Unauthorized();
        }
        private async Task<bool> CheckPassword(Credentials creds) {
            IdentityUser user = await userManager.FindByNameAsync(creds.Username);
            if (user != null) {
                foreach (IPasswordValidator<IdentityUser> v in
                         userManager.PasswordValidators) {
                    if ((await v.ValidateAsync(userManager, user,
                            creds.Password)).Succeeded) {
                        return true;
                    }
                }
            }
            return false;
        }

        public class Credentials {
            [Required]
            public string Username { get; set; }
            [Required]
            public string Password { get; set; }
        }
    }
}
```

The UserManager<T> class defines a PasswordValidators property that returns a sequence of objects that implement the IPasswordValidator<T> interface. When the Token action method is invoked, it passes the credentials to the CheckPassword method, which enumerates the IPasswordValidator<T> objects to invoke the ValidateAsync method on each of them. If the password is validated by any of the validators, then the Token method creates a token.

The JWT specification defines a general-purpose token that can be used more broadly than identifying users in HTTP requests, and many of the options that are available are not required for this example. The token that is created in Listing 39-27 contains a payload like this:

```
...
{
  "unique_name": "bob",
  "nbf": 1579765454,
  "exp": 1579851854,
  "iat": 1579765454
}
...
```

The unique_name property contains the name of the user and is used to authenticate requests that contain the token. The other payload properties are timestamps, which I do not use.

The payload is encrypted using the key defined in Listing 39-27 and returned to the client as a JSON-encoded response that looks like this:

```
...
{
    "success":true,
    "token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
...
```

I have shown just the first part of the token because they are long strings and it is the structure of the response that is important. The client receives the token and includes it in future requests using the Authorization header, like this:

```
...
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
...
```

The server receives the token, decrypts it using the key, and authenticates the request using the value of the `unique_name` property from the token payload. No further validation is performed, and requests with a valid token will be authenticated using whatever username is contained in the payload.

Authenticating with Tokens

The next step is to configure the application to receive and validate the tokens, as shown in Listing 39-28.

Listing 39-28. Authenticating Tokens in the Startup.cs File in the Advanced Project

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Advanced.Models;
using Microsoft.AspNetCore.ResponseCompression;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using System.Security.Claims;
using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace Advanced {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:PeopleConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });
            services.AddControllersWithViews().AddRazorRuntimeCompilation();
            services.AddRazorPages().AddRazorRuntimeCompilation();
            services.AddServerSideBlazor();
            services.AddSingleton<Services.ToggleService>();

            services.AddResponseCompression(opts => {
                opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
                    new[] { "application/octet-stream" });
            });
            services.AddDbContext<IdentityContext>(opts =>
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:IdentityConnection"]));

            services.AddIdentity<IdentityUser, IdentityRole>()
                .AddEntityFrameworkStores<IdentityContext>();

            services.Configure<IdentityOptions>(opts => {
                opts.Password.RequiredLength = 6;
                opts.Password.RequireNonAlphanumeric = false;
                opts.Password.RequireLowercase = false;
                opts.Password.RequireUppercase = false;
                opts.Password.RequireDigit = false;
                opts.User.RequireUniqueEmail = true;
                opts.User.AllowedUserNameCharacters = "abcdefghijklmnopqrstuvwxyz";
            });

            services.AddAuthentication(opts => {
                opts.DefaultScheme =
                    CookieAuthenticationDefaults.AuthenticationScheme;
                opts.DefaultChallengeScheme =
                    CookieAuthenticationDefaults.AuthenticationScheme;
            }).AddCookie(opts => {
                opts.Events.DisableRedirectForPath(e => e.OnRedirectToLogin,
                    "/api", StatusCodes.Status401Unauthorized);
                opts.Events.DisableRedirectForPath(e => e.OnRedirectToAccessDenied,
                    "/api", StatusCodes.Status403Forbidden);
            }).AddJwtBearer(opts => {
                opts.RequireHttpsMetadata = false;
                opts.SaveToken = true;
                opts.TokenValidationParameters = new TokenValidationParameters {
                    ValidateIssuerSigningKey = true,
                    IssuerSigningKey = new SymmetricSecurityKey(
                        Encoding.ASCII.GetBytes(Configuration["jwtSecret"])),
                    ValidateAudience = false,
                    ValidateIssuer = false
                };
                opts.Events = new JwtBearerEvents {
                    OnTokenValidated = async ctx => {
                        var usrmgr = ctx.HttpContext.RequestServices
                            .GetRequiredService<UserManager<IdentityUser>>();
                        var signinmgr = ctx.HttpContext.RequestServices
                            .GetRequiredService<SignInManager<IdentityUser>>();
                        string username =
                            ctx.Principal.FindFirst(ClaimTypes.Name)?.Value;
                        IdentityUser idUser = await usrmgr.FindByNameAsync(username);
                        ctx.Principal =
                            await signinmgr.CreateUserPrincipalAsync(idUser);
                    }
                };
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {

            // ...statements omitted for brevity...
        }
    }
}
```

The AddJwtBearer adds support for JWT to the authentication system and provides the settings required to decrypt tokens. I have added a handler for the OnTokenValidated event, which is triggered when a token is validated so that I can query the user database and associate the IdentityUser object with the request. This acts as a bridge between the JWT tokens and the ASP.NET Core Identity data, ensuring that features like role-based authorization work seamlessly.

Restricting Access with Tokens

To allow a restricted endpoint to be accessed with tokens, I have modified the Authorize attribute applied to the Data controller, as shown in Listing 39-29.

Listing 39-29. Enabling Tokens in the DataController.cs File in the Controllers Folder of the Advanced Project

```
using Advanced.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;

namespace Advanced.Controllers {

    [ApiController]
    [Route("/api/people")]
    [Authorize(AuthenticationSchemes = "Identity.Application, Bearer")]
    public class DataController : ControllerBase {
        private DataContext context;

        // ...methods omitted for brevity...
    }
}
```

The AuthenticationSchemes argument is used to specify the types of authentication that can be used to authorize access to the controller. In this case, I have specified that the default cookie authentication and the new bearer tokens can be used.

Using Tokens to Request Data

The final step is to update the JavaScript client so that it obtains a token and includes it in requests for data, as shown in Listing 39-30.

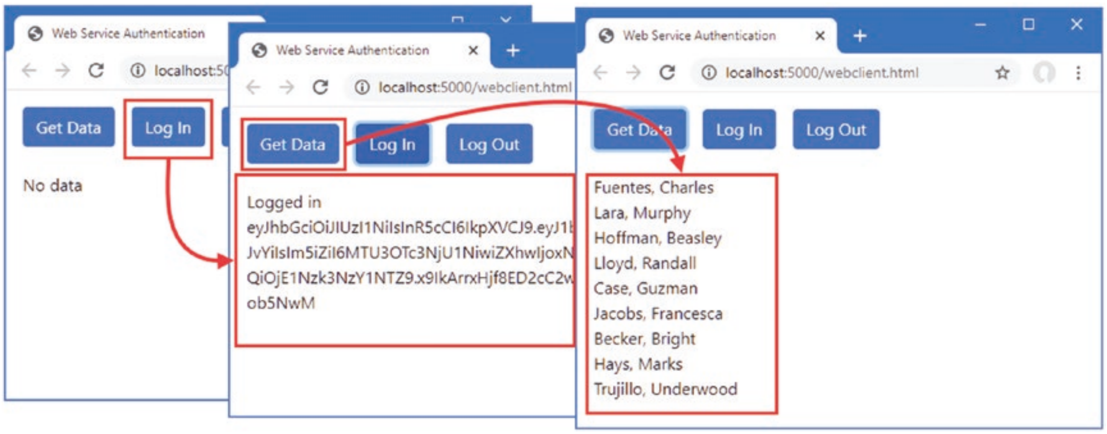
Listing 39-30. Using Tokens in the webclient.js File in the wwwroot Folder of the Advanced Project

```
const username = "bob";
const password = "secret";
let token;

window.addEventListener("DOMContentLoaded", () => {
    const controlDiv = document.getElementById("controls");
    createButton(controlDiv, "Get Data", getData);
    createButton(controlDiv, "Log In", login);
    createButton(controlDiv, "Log Out", logout);
});

async function login() {
    let response = await fetch("/api/account/token", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ username: username, password: password })
    });
    if (response.ok) {
        token = (await response.json()).token;
        displayData("Logged in", token);
    } else {
        displayData(`Error: ${response.status}: ${response.statusText}`);
    }
}

async function logout() {
    token = "";
    displayData("Logged out");
}

async function getData() {
    let response = await fetch("/api/people", {
        headers: { "Authorization": `Bearer ${token}` }
    });
    if (response.ok) {
        let jsonData = await response.json();
        displayData(...jsonData.map(item => `${item.surname}, ${item.firstname}`));
    } else {
        displayData(`Error: ${response.status}: ${response.statusText}`);
    }
}

function displayData(...items) {
    const dataDiv = document.getElementById("data");
    dataDiv.innerHTML = "";
    items.forEach(item => {
        const itemDiv = document.createElement("div");
        itemDiv.innerText = item;
        itemDiv.style.wordWrap = "break-word";
        dataDiv.appendChild(itemDiv);
    })
}

function createButton(parent, label, handler) {
    const button = document.createElement("button");
    button.classList.add("btn", "btn-primary",  "m-2");
    button.innerText = label;
    button.onclick = handler;
    parent.appendChild(button);
}
```

The client receives the authentication response and assigns the token so it can be used by the `GetData` method, which sets the `Authorization` header. Notice that no logout request is required, and the variable used to store the token is simply reset when the user clicks the Log Out button.

■ **Caution** It is easy to end up authenticating with a cookie when attempting to test tokens. Make sure you clear your browser cookies before testing this feature to ensure that cookies from previous tests are not used.

Restart ASP.NET Core and request `http://localhost:5000/webclient.html`. Click the Log In button, and a token will be generated and displayed. Click the Get Data button, and the token will be sent to the server and used to authenticate the user, producing the results shown in Figure 39-13.



No data

Logged in
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1b
JvYiIsIm5iZiI6MTU3OTc3NjU1NiwiZXhwIjoxN
QiOjE1Nzk3NzY1NTZ9.x9IkArrxHjf8ED2cC2w
ob5NwM

Fuentes, Charles
Lara, Murphy
Hoffman, Beasley
Lloyd, Randall
Case, Guzman
Jacobs, Francesca
Becker, Bright
Hays, Marks
Trujillo, Underwood

Figure 39-13. *Using a token for authentication*

Summary

In this chapter, I showed you how to apply authentication and authorization in an ASP.NET Core application. I explained the process for authenticating users and restricting access to endpoints. I explained how users are authorized in Blazor applications, and I demonstrated how web service clients can be authenticated using cookies and bearer tokens.

That's all I have to teach you about ASP.NET Core. I can only hope that you have enjoyed reading this book as much as I enjoyed writing it, and I wish you every success in your ASP.NET Core projects.

Index